



National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services Branch

Direction des acquisitions et
des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file - Votre référence

Our file - Notre référence

NOTICE

AVIS

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

If pages are missing, contact the university which granted the degree.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

MODELLING AUTOMOTIVE ENGINES FOR AUTOMATED DIAGNOSIS

by

Charles David Hunter

B.A.Sc., The University of British Columbia, 1984

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE

in the School
of
Engineering Science

© Charles David Hunter 1991
SIMON FRASER UNIVERSITY
November 1991

All rights reserved. This thesis may not be
reproduced in whole or in part, by photocopying
or by other means, without the permission of the author.



National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services Branch

Direction des acquisitions et
des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file *Votre référence*

Our file *Notre référence*

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-83676-8

Canada

Approval

Name: Charles David Hunter

Degree: Master of Applied Science

Title of Thesis: Modelling Automotive Engines for
Automated Diagnosis

Examining Committee: Dr. John C. Dill, Chairperson

Dr. John D. Jones
Senior Supervisor

Dr. William S. Havens
Supervisor

Mr. Stefan W. Joseph
External Examiner

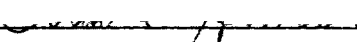
Date Approved: Dec. 6. 1991

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

Modelling Automotive Engines for Automated Diagnosis

Author: 
(signature)

Charles D. Hunter
(name)

December 11, 1991
(date)

Abstract

As the complexity of vehicular electronic control systems increases, automobiles are becoming increasingly difficult to diagnose. Diagnostic expert systems based on production rules have been used with limited success for small, well understood domains. Model-based diagnostic systems offer advanced capabilities, as displayed for domains such as electronic circuits, where efficient models are easily developed. The success of model-based technology for automotive diagnosis depends on the availability of efficient automotive diagnostic models.

Engine models are presently developed for two purposes, design and control. Models for design often require extensive computation, and deal with variables unrelated to diagnosis. Control models require empirical results from lengthy bench testing. Models specifically for engine diagnosis have not been reported.

All models are necessarily incomplete, and even the most detailed models will be unable to find all diagnoses. Quantitative models pursue excessively detailed calculations. Qualitative models are potentially more efficient while still providing the necessary detail for diagnosis. Our model represents physical components as primitives, and groups of components working together as composite components. We incorporate a specialization hierarchy, which uses inheritance to centralize, and reduce the storage of, information that is common to similar types of components. We also utilize a composition hierarchy to derive the structure and behaviour of complex systems from that of its sub-components.

We present a prototype engine sub-system model to diagnose single, non-intermittent faults, implemented with the Echidna constraint reasoning system, which incorporates constraint logic programming, truth maintenance, and dependency backtracking, all in an object-oriented framework. Performance of the prototype is reported, and is extrapolated to estimate the

performance of a complete engine model. Limitations of the prototype model, and suggestions for further research, are discussed.

to my family, for not rushing me

George, Pat, Margaret, Tim, Sarah

Acknowledgements

I would like to thank the director of the Simon Fraser University Expert Systems Lab Dr. William S. Havens for leading the diagnosis group, acting as second supervisor, and for his determined generation of, and participation in, academic debate, the graduate students of the diagnosis research group, Afwarman Manaf, for his support and encouragement, and Peter MacDonald, for forcing us to broaden our research horizons, McCarney Technologies Inc., for their financial support, and Stefan Joseph for cheerfully acting as McCarney liaison and external examiner.

It was a pleasure to work with the staff of the Expert Systems Lab - Miron Cuperman, Rod Davison, and especially Sue Sidebottom, whose programming assistance and patience were invaluable.

Finally, my deepest thanks to my supervisor Dr. John Dewey Jones, for his confidence, for always being available, for prodding me when I needed it, for always listening, and for the many revisions of this thesis he enthusiastically proof read.

Table of Contents

Approval	ii
Abstract	iii
Acknowledgements	vi
List of Tables	x
List of Figures	x
1. Introduction	1
1.1. Automotive Technology.....	2
1.2. Automated Reasoning by Expert Systems	4
1.2.2. Rule-Based Expert Systems	4
1.2.3. Model-Based Expert Systems	6
1.3. Motivation.....	7
2. Literature Review	8
2.1. Automotive Diagnostic Expert Systems.....	8
2.2. Model-Based Diagnosis.....	8
2.3. Automotive Engine Modelling	14
2.3.1. Modelling for Design.....	14
2.3.2. Modelling for Control.....	15
2.3.3. Failure Detection and Isolation.....	17
3. Automated Model-Based Diagnosis	18
3.1. Symptom Generation.....	20
3.2. Candidate Set Generation.....	22
3.2.1. Naive Search.....	24
3.2.2. Abductive-Rule-Based Search.....	25

3.2.3. Probabilistic Search	28
3.2.3.1. Global Probabilistic Strategy	28
3.2.3.2. Local Probabilistic Strategy	29
3.3. Discriminating Between Candidates.....	30
4. Engine Modelling for Diagnosis	33
4.1. Completeness.....	33
4.2. Competence.....	35
4.3. Qualitative and Quantitative Models.....	36
4.4. Assumptions and Limitations	36
4.4.1. Time - Steady State, Intermittent Faults	37
4.4.2. Single Fault Assumption	38
4.4.3. Correct configuration of components.....	40
4.4.4. Measurements and tolerances.....	40
4.5. Component Models.....	40
4.5.1. Primitive Component Models.....	41
4.5.2. Composite Component Models.....	44
4.5.2.1. System Component Models.....	46
4.5.2.2. Compound Component Models.....	49
4.6. Hierarchical Composition.....	52
4.6.1. Specialization Hierarchy.....	52
4.6.2. Composition Hierarchies	54
4.7. Suggestions for Future Work on Modelling.....	55
4.7.1. Ranking Candidates.....	55
4.7.2. Discrimination between Candidates.....	56
4.7.2. Time.....	56

5. Prototype Diagnostic Model	57
5.1. Echidna Constraint Reasoning System	57
5.2. Prototype Vehicle.....	58
5.3. Prototype Sub-systems.....	59
5.4. Simulated Diagnosis.....	63
5.4.1. Analysis of Performance.....	69
5.5. Suggested Implementation Improvements.....	70
6. Conclusions	73
Appendix A - Knowledge Bases	77
A.1. Component Knowledge Base.....	77
A.2. Primitive Knowledge Base	86
A.3. Compound Knowledge Base.....	111
A.4. System Knowledge Base.....	114
Appendix B - Simulated Diagnostic Sessions	128
B.1. Power System Diagnostic Simulation.....	128
B.1.1. Input Data-Base File.....	128
B.1.2. Output File.....	130
B.2. Canister Purge System Diagnostic Simulation	134
B.2.1. Input Data-Base File.....	134
B.2.2. Output File.....	136
References	144

List of Tables

Table 5-1: Number of Components.....	65
Table 5-2: Power System Performance.....	66
Table 5-3: CCP System Performance.....	67
Table 5-4: Engine Performance.....	69
Table 5-5: Vehicle Performance.....	69

List of Figures

Figure 1-1: Automobile Diagnostic Expert System.....	6
Figure 3-1: Diagnostic Procedure.....	19
Figure 3-2: Symptom Generation.....	21
Figure 3-3: Candidate Generation.....	23
Figure 3-4: Abductive Candidate Generation.....	26
Figure 4-1: Primitive Component Model - Simple Switch.....	42
Figure 4-2: Composite System Component - twoSwitches diagram.....	48
Figure 4-3: Composite Compound Component - ignitionSwitch.....	51
Figure 4-4: Partial Vehicle Specialization Hierarchy.....	53
Figure 4-5: Partial Vehicle Composition Hierarchy.....	54
Figure 5-1: Prototype Model Composition Hierarchy.....	60
Figure 5-2: 12-Volt Power Distribution System.....	61
Figure 5-3: Charcoal Canister Purge System.....	62
Figure 5-4: Fuel System.....	63

1. Introduction

Given tests, measurements, and observations of a malfunctioning device, diagnosis is the task of identifying the faulty component(s) which caused the malfunction(s). The objective is to develop a modelling strategy suitable for the efficient and accurate model-based automated diagnosis of complex electrical/mechanical devices, and to use this strategy to develop a prototype diagnostic model for a sub-system of an electronically controlled automobile engine. The prototype model must be extendible to the diagnosis of complete vehicles from a wide range of automobile manufacturers, models, ages, and locations, and must serve as the basis for commercial implementation. Specific goals of this thesis are to:

- A. Evaluate the current state and future trends in automobile diagnosis and diagnostic tools, especially relating to electronic and computerized control systems.
- B. Evaluate the diagnostic suitability of current mathematical and computational automobile engine models.
- C. Study current literature on automated model-based diagnosis.
- D. Develop a prototype diagnostic model and test it with the Echidna constraint reasoning system [Sidebottom 91], an expert system shell which incorporates model-based reasoning, logic programming, constraint propagation, and hypothetical reasoning.

The remainder of this chapter is concerned with goal A above. Chapter 2 deals with goals B and C by reviewing three research areas - automotive diagnostic expert systems, model-based diagnosis, and automotive engine modelling.

Chapters 3 to 5 deal with goal D. Chapter 3 describes our general approach to automated model-based diagnosis, providing a foundation for the following chapters.

Using simple examples from the automotive domain chapter 4 develops the structure and knowledge representation of our prototype diagnostic model. It is assumed the reader has read chapter 3, or is familiar with the concepts and definitions presented there.

Chapter 5 presents the prototype automobile diagnostic model, preceded by a quick description of the Echidna Constraint Reasoning System developed at Simon Fraser University's Center for Systems Science [Sidebottom 91]. Echidna's syntax is based on that of Prolog. Readers not familiar with Prolog could consult [Bratko 86] or [Sterling 86]. The performance of the Echidna implemented model, and obvious improvements to that implementation, are discussed. Echidna code for our model, and input files and output data from diagnostic simulations, is shown in the appendices.

1.1. Automotive Technology

Since the early eighties all automobiles have had some form of electronic control, development being driven by stricter U.S. government environmental emissions and fuel consumption regulations for passenger vehicles. Emission regulations specified maximum allowable emissions for a specific test cycle simulating city and highway driving including idling, accelerating, decelerating, and cruising, and corporate fuel consumption averages and penalties for exceeding them were set. Initially the controlled inputs were fuel flow rate, spark timing, and exhaust gas recirculation, and the optimized outputs were exhaust and evaporative emissions, and fuel consumption. Electronic control systems consisted of a microcomputer (often called an electronic control module or ECM) and a number of sensors and actuators.

The electronic revolution swept through the automobile manufacturers. The relative ease of designing and developing electronic components and the existence of on-board hardware led to a rapid expansion of electronic control to other areas of the engine and vehicle. Performance was a valuable marketing feature and manufacturers found that their vehicle's performance could be enhanced by altering the control strategy when the vehicle was not at an operating point specified in the government test cycle. While continuing to meet increasingly stringent regulations for emissions (and, due to changes in government policy, somewhat neglecting fuel consumption) some manufacturers are now attempting to improve performance, driveability, and driver/passenger comfort. Electronic components are replacing their mechanical equivalents resulting in reduced volume, mass, maintenance, and cost, increased reliability, smoother operation, and fewer moving parts and wear surfaces. Each new model and year introduces new electronic components, sensors, and control strategies. Emission requirements will soon be extended to cover trucks and other commercial vehicles.

As electronic control systems become more complicated and extensive, it is increasingly difficult to diagnose and repair engine and vehicle faults. Electronic control systems provide excellent reliability, but when they fail today's automobile technician can not be expected to diagnose them using yesterday's (non-computerized) tools.

Diagnosis procedures are supplied by the manufacturers, but are generic and cover only a small fraction of possible malfunctions. The ECM stores and displays trouble codes, (which are unique to each manufacturer), but these are set conservatively to minimize false alarms. While manufacturers have supplied their dealer service facilities with sophisticated electronic tools to aid diagnosis, (with varying degrees of success), the average service station is quickly becoming unable to repair its customers' vehicles efficiently and accurately. A selection of third-party generic electronic diagnostic tools is available [Joseph 89] that display the outputs and inputs of the electronic control module in either graphic, tabular or text form. While these tools are of

great assistance, the actual diagnosis is performed by the technician, requiring him to have complete knowledge of the system's operation.

1.2. Automated Reasoning by Expert Systems

Expert systems are computer programs which use specialized databases (known as knowledge bases) to reason about some limited domain. The two primary components of an expert system are the inference engine and the knowledge base. When given a query, the inference engine consults the knowledge base to produce an answer. The separation between the knowledge base and the inference engine distinguishes expert systems from conventional programs, which combine the knowledge and control into algorithm(s) which compute answers. Expert systems have been classified into the following applications: diagnosis, interpretation, monitoring, debugging, prediction, design, planning, repair, instruction, and control, [Hayes-Roth 83] which can in turn be grouped under two major headings - analysis, which includes diagnosis, and synthesis, which, for example, includes design. Diagnosis is one of the simpler applications because we usually know much about the object we are diagnosing. In design we do not know the form of the artifact until after we are finished. Diagnosis is therefore a better choice for expanding expert systems into large, complex domains.

1.2.2. Rule-Based Expert Systems

The first generation of expert systems relied on knowledge bases representing knowledge only as production rules of the form IF <antecedent> THEN <consequent>. It was shown that expert level performance could be obtained if the domain was sufficiently restricted, and the underlying complexities of the real-world problem were replaced by domain-specific rules obtained from an expert [Buchanan 84]. Primary examples of rule-based expert systems are

MYCIN (blood disorder diagnosis) [Buchanan 84], MACSYMA (symbolic mathematics) [Martin 71], PROSPECTOR (mineral exploration) [Duda 79], and DOMINIC (engineering design) [Howe 86].

Although expert systems and expert system shells ("empty" expert systems ready for domain-specific knowledge) have been given greater computational power and a wide range of programming, reasoning, and knowledge acquisition tools [KEE 86][G2 88], they still depend on the production-rule knowledge representation. Common criticisms of rule-based systems are that they have no meta-knowledge (e.g. do not know their limits and do not exhibit "graceful degradation" at their limits), they are difficult to organize, update, and maintain when the number of rules becomes large, and most importantly, they have no knowledge of the domain's composition (the set of components which comprise the domain), structure (the way those components are connected), and behaviour (the way those components behave and interact).

Rule-based expert systems are generally recommended and successful only when the given problem domain is small and well-understood, and when there is general agreement among domain experts [Luger 89]. Yet those domains which may most benefit from expert systems, for example diagnosis of electronic circuits, medical disorders, and automobiles, do not now, and may never, fit this description. Thus, while there has been extensive work in these areas, there are many problems still to be solved.

Many elementary AI textbooks use automotive diagnosis for a sample domain [Luger 89]. Invariably the example uses a production-rule knowledge representation, and usually involves a simplified version of the electrical or charging systems. Although popular as an example, no one has yet been able to develop a diagnostic expert system that will significantly benefit an automotive technician. Since we design automobiles, we understand their composition, structure, and behaviour. And yet, current rule-based expert system technology has been unable to provide much more than textbook examples. Moreover,

automobile electrical systems are no longer simple, especially with the advent of computerized control systems.

1.2.3. Model-Based Expert Systems

Model-based expert systems use a model (a behavioural theory about some class of artifact, situation, concept) instead of a collection of condition/action pairs obtained from a domain expert. Model-based systems attempt to capture structure and behaviour explicitly [Davis 84]. A model-based diagnostic system for a mechanical device, for example, is developed directly from the device's design description. Just as in rule-based expert systems, model-based systems maintain the separation between the knowledge base and the inference engine.

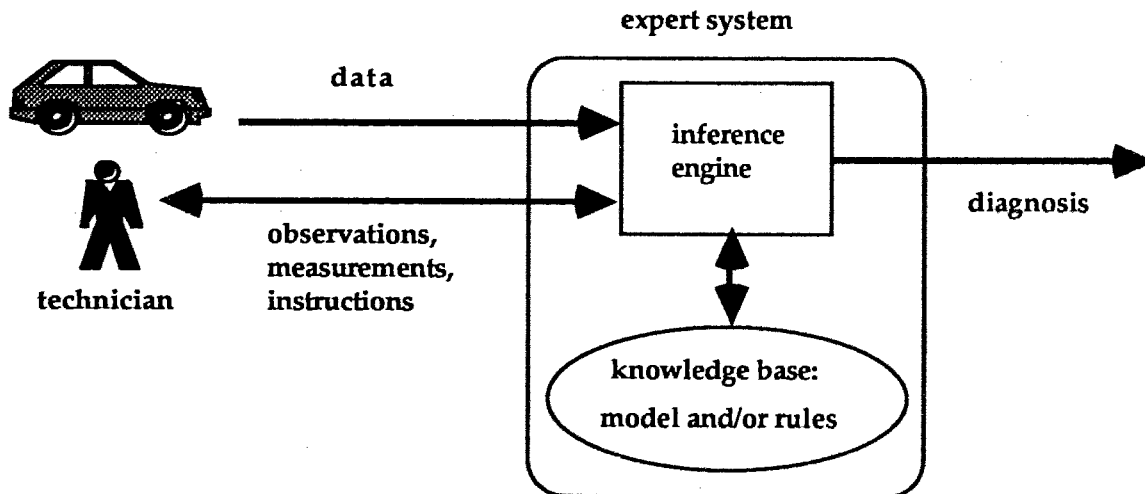


Figure 1-1
Automobile Diagnostic Expert System

The potential advantages of model-based systems over rule-based systems are many [Davis 84] [Fink 85] [Hamscher 87]. The model has a structure resulting naturally from the structure of the artifact being modelled, and can take advantage of similarities between different components and assemblies. Maintenance of model-based expert systems is easier, as changes to the artifact

can naturally be included as changes to the model. While development and verification of model-based diagnostic systems does not require as much of a diagnostician's time, it does require more time from an engineer or designer, but, in general, the designer/engineer will be better able to formalize his knowledge. Perhaps the most important is that a model can be used for more than one task (i.e., recognition, explanation, prediction) [Genesereth 84]. For example, a rule-based diagnostic system can not be used to identify whether a particular artifact belongs in its domain, nor can it make any predictions about artifacts within its domain.

1.3. Motivation

Our prototype diagnostic model will be designed for an expert system that can be extended into a commercial product covering a wide range of vehicles and/or automotive sub-systems. The result of this commercial enterprise will be the improved diagnostic ability of automotive technicians, resulting in efficient and accurate automobile repairs, and happy automobile owners and manufacturers' warranty departments. The reduction in malfunctioning (and undiagnosed) emission-control systems will reduce the automobile's contribution to environmental decay. Fuel consumption will decrease. Automobile manufacturers will be comfortable designing more sophisticated, efficient, and safe vehicles, knowing that they can be effectively diagnosed. The application of electronic control to trucks and other engines can be handled efficiently.

Finally, it is believed that the modelling techniques developed can be applied to the diagnosis of other complex devices and domains.

2. Literature Review

This chapter provides a limited overview of three research areas related to our study - automotive diagnostic expert systems, model-based diagnosis, and automotive engine modelling. The vast quantity of publications covering these topics prohibits a complete review, but this review is representative of current work and provides a firm foundation for further study.

2.1. Automotive Diagnostic Expert Systems

The application of model-based diagnosis to automotive engines, and mechanical systems in general, has not been widely reported. Prototype systems that use shallow and deep knowledge bases have been developed and tested on a gas heating system [Fink 85], and dynamic systems [Abu-Hanna 88].

A number of authors have developed prototypes of rule-based automobile diagnosis expert systems. "Problem-cause" pairs were employed to diagnose Nissan's Electronic Concentrated Engine Control System [Tomikasi 87], and a commercially developed expert system shell was tested on '81-'83 General Motors computer-controlled carbureted engines [Klausmeier 86].

2.2. Model-Based Diagnosis

The most popular domain for the study of model-based diagnosis is electronic circuits, for a number of reasons. Electronic circuit diagnosis is a real problem due to continually increasing complexity in electronic circuit design. Most, if not all, researchers have a background in electronics or computers and are therefore intimately familiar with the domain. Using a common domain

allows easy comparison of results. Most importantly, however, the electronic circuit domain is blessed with a close relationship between structure and function. Knowing a circuit's structure (i.e., the arrangement of resistors, capacitors, transistors, etc.) the function of the circuit can easily be ascertained and a reasonably accurate model of behaviour can be developed relatively easily. The ease of modelling electronic circuit behaviour allows researchers to concentrate on diagnostic theory rather than complexities in modelling. Still, there is presently no unified theory capable of diagnosing complex and time-dependent devices efficiently [Hamscher 90b], because traditional circuit models do not explicitly represent aspects of the device a diagnostician would consider, such as the shape of signals (rising, falling, steady, oscillating, etc.), and the kinds of failures that are likely to occur.

Early model-based expert systems dealt with path sensitization (hypothesizing a component in a particular faulty state, backward propagating to measurable inputs, and forward propagating to measurable outputs), hierarchical description, and qualitative properties of signals (i.e., smooth, spiky, random, rising, decreasing, etc.) as well as their values were discussed [Shirley 83]. Genesereth presented diagnosis based on design descriptions, not symptom/fault rules [Genesereth 84], building upon his earlier work on diagnosis using hierarchical design models [Genesereth 81]. Davis wrote on diagnosis from structure and behaviour [Davis 84], and recently co-authored a survey on model-based troubleshooting [Davis 88]. Reiter set the foundation for a large body of future work by formalizing diagnosis using first-order logic [Reiter 87]. Hamscher added to this with work on diagnosis from first principles [Hamscher 87], and Dague tackled the difficulties in modelling for troubleshooting [Dague 87].

A system called the "General Diagnostic Engine" (GDE) [deKleer 87], based on the attractive premise that only correct modes of operation need be explicitly represented, was implemented and tested on electronic circuits. It was able to diagnose multiple faults, represented diagnoses as minimal sets of violated assumptions, used an incremental diagnostic procedure, was domain

independent, and combined model-based prediction with sequential diagnosis to propose measurements to localize faults. GDE relied heavily on three important topics described below: reason maintenance, Bayesian probability methods, and entropy.

Reason maintenance systems (RMSs) [Doyle 79], sometimes called truth maintenance systems (TMSs), were developed to support non-monotonic reasoning. In monotonic reasoning new beliefs are derived from old beliefs, old beliefs never change, and the number of beliefs is ever increasing. Non-monotonic reasoning more closely matches human reasoning by allowing beliefs based on incomplete information, or other beliefs. When new information forces a modification of a previously held belief, all beliefs depending on that belief may also have to be modified. The TMS maintains a record of the presently held beliefs, and the reasons for those beliefs, which may be other beliefs. When a reason for a belief is removed, that belief, and all beliefs depending on that belief, may have to be removed (if there are no other reasons to believe them). The TMS promotes data-dependency backtracking, where only those choices directly responsible for a contradiction are changed. This contrasts with chronological backtracking commonly used in Prolog, where a contradiction results in a backwards chronological change of choices, whether or not those choices directly contributed to the contradiction.

Many TMSs (e.g., Doyle's Justification Truth Maintenance System, or JTMS) support only one solution at a time (i.e., a single context), but many problems require computing and comparing multiple solutions. The assumption-based TMS (ATMS) [deKleer 86a,b] is a multiple-context TMS developed precisely for these problems. The ATMS can in parallel investigate all contexts supported by different sets of assumptions, without confusion, and without re-derivation of shared intermediate conclusions. Less backtracking is required than in a TMS, and minimal set covers are utilized to reduce data storage.

Bayesian methods allow diagnostic reasoning under uncertainty by providing a formalism for calculating the probabilities we need by combining and

manipulating the probabilities that are easiest to estimate [Pearl 88]. When there is more than one diagnosis that fits the evidence it is desirable to know the probabilities of those diagnoses. However, the probabilities of competing diagnoses depend on the evidence that has been collected. Some probabilities can be estimated fairly easily and stored for later use (for instance, the probability that a certain component of a particular engine model will fail, and the probability that a particular component failure will result in particular symptoms), but it is impossible to tabulate in advance the probabilities of each diagnosis relative to all possible combinations of evidence. Bayes law can be interpreted as:

$$P(D|e) = P(e|D) P(D) / P(e)$$

where D is a particular diagnosis, e is the evidence gathered so far, $P(D|e)$ is the *posterior* probability of the diagnosis D given the evidence e , $P(e|D)$ is the conditional probability that the malfunction explained by diagnosis D will create the evidence e , $P(D)$ is the *prior* probability of the diagnosis, and $P(e)$ is a normalizing factor that can be approximated by the sum of all prior probabilities (i.e., the sum over all D 's of $P(e|D)$) [Pearl 88]. $P(e|D)$ and $P(D)$ can be estimated and stored in the diagnostic system. Posterior probabilities are used to rank competing diagnoses.

GDE used a one-step look-ahead strategy based on Shannon entropy [Shannon 49] [Papoulis 84]. Entropy is a measure of the amount of information left in an information source. The entropy of an information source is defined as:

$$H(Z) = - \sum_z P(z) \ln P(z)$$

where Z is a variable measured at the source with probability distribution $P(z)$, and $H(Z)$ is the entropy of Z . When all possible messages from a source (i.e., all z 's) have equal probability entropy is at a maximum (much information remains) because the probability distribution gives little hint as to which

message you might receive. The entropy is small when the probability of one message greatly exceeds that of the others, so measuring that source provides less information as it will most likely yield the message with the high probability. In the limit where the probability of a particular message is 1 and the probability of all others is 0, that source can only give one message, so there is no information left in the source.

In a diagnostic scheme each possible measurement point is an information source. Measurements are expensive, so it is desirable to minimize the number of measurements needed to find the correct diagnosis. Entropy can be used to estimate which measurement will leave the least information in the system (i.e., eliminate the largest number of possible diagnoses). The expected entropy after the next measurement can be calculated for each possible measurement point as a weighted sum of the expected entropies for each possible value of that measurement point. The measurement point with the lowest expected entropy after the next measurement will on average minimize the number of measurements needed to isolate the correct diagnosis.

GDE's lack of explicit failure modes unfortunately resulted in physically impossible failures being considered and presented as possible diagnoses. For example, a bulb could fail by lighting when no voltage was applied. deKleer modified his GDE with the addition of behavioural modes in a system called SHERLOCK [deKleer 89]. Here, a component can have an arbitrary number of behavioural modes, some representing correct operation, some faulty operation, and one unknown. Bayesian probability calculations based on the likelihood of each behavioural mode were used to handle the combinatorial explosion already plaguing GDE.

Hamscher also modified GDE, but instead incorporated two distinct hierarchies, one physical and the other functional [Hamscher 90a]. Although fault models were included, they were not required, and were only used heuristically to guide the search and limit the size of the search space. His system was named the "Extended Diagnostic Engine", or XDE.

Struss provided yet another approach in a system called GDE+ [Struss 89]. Here, explicit use of fault modes was incorporated. It also added to GDE by proving the correctness of components, and ruling out implausible hypotheses, and resulted in an extended version of the ATMS [Struss 83].

Early work in the domain of medical diagnosis used a hierarchical structure of disorders (i.e., problems, syndromes, diseases) with a strong dependence on Bayesian probabilities [Ben-Bassat 80]. Peng and Reggia introduced another interpretation of Bayesian classification theory [Peng 86], and parsimonious covering theory based on causal associations between disorders and manifestations (symptoms), where a parsimonious or "simple" cover is a set of possible disorders which satisfies all known information and meets some criterion for being "simple". Examples of parsimony criteria are: 1) single - disorder diagnoses only, 2) minimality, where each diagnosis contains the smallest number of disorders needed to cover all symptoms, 3) irredundancy where no proper subset of a diagnosis is also a diagnosis, and 4) relevancy, where each disorder in a diagnosis predicts at least one of the symptoms.

A review of hypothetical reasoning based on abduction [Goebel 90] explains and distinguishes several forms of inference and reasoning central to diagnosis. Abductive reasoning is an unsound rule of inference where one hypothesis (perhaps from a set of possible hypotheses) is chosen that explains or accounts for all known observations. In automobile diagnosis this corresponds to reasoning from symptoms and engine behaviour to a hypothetical diagnosis of broken components. This contrasts with induction, another unsound rule of inference, which reasons from symptoms and broken components to hypothetical engine behaviour, and deduction, a sound rule of inference which reasons from broken components and engine behaviour to symptoms. A survey of abductive reasoning in multiple fault diagnosis [Finin 89] summarizes current research into five different approaches, concluding that the emerging consensus is integrating parsimonious set covering or logic formalism (which can be transformed from each other), causal models to

incorporate intermediate pathological states, and Bayesian probabilities freed of most restrictions on independence.

2.3. Automotive Engine Modelling

Unlike electronic circuits, the behaviour of mechanical systems is not easily derived from their structure, so a single mechanical system may be represented by a variety of models each developed and tuned for a different purpose. Engine modelling is presently pursued for two different reasons - engine design and engine control. These models are not developed with expert systems implementation in mind. The REPAIR project [Lee 90] has recently explored the use of qualitative reasoning for model-based diagnosis of simple mechanical devices. There is presently no literature available on models specifically for engine diagnosis.

2.3.1. Modelling for Design

Models for engine design simulate the dynamic and quasi - static performance of an engine, allowing the designer to predict the outcome of a particular design without having to complete costly prototype construction, testing, and analysis. Some of these models include not only physical properties and quantities but also the effect of part geometries (i.e., combustion chamber shape), and materials.

Diagnosis covers only those variables which can be changed after the vehicle is in operation (i.e., calibration, adjustments, replacement of components, etc.), while engine design models are concerned with variables that can be changed before engine components are made (i.e., valve seat angles, displacement, number of cylinders, etc.).

Engine design models of vastly different size have been developed - some requiring hours of mainframe cpu time, and others designed for personal computers [Morel 88] [Blumberg 79]. The cost and time of a computed diagnosis must be less than that of the a technician, so efficiency in terms of hardware and time are critical. In the short term it is unlikely that automobile service centers will have access to hardware more powerful than a PC.

2.3.2. Modelling for Control

A recent comprehensive survey of internal combustion engine models for control system design [Powell 87] defined models by two parameters, the first dealing with time (steady state, or dynamic), and the second dealing with the use of physics (input/output or physical). While there are no true steady-state processes in a reciprocating engine, the steady-state models describe the condition where air flows, fuel flows, and engine temperatures have reached a steady, but oscillating state. I/O models are developed solely by matching outputs with their causal inputs, while physical models are derived from underlying physical principles. Of course, engine models for control need not consider part geometry or material.

Much of the early work on dynamic physical models was done on diesel engines, but many of the results have direct application to spark ignition (SI) engines. During the 1970's research concentrated on I/O models, because stringent emission regulations and demand for fuel economy made it important to develop models and controllers quickly. Although focus later shifted to physical models, all models still suffer a significant reliance on engine testing for empirical parameters.

Early research on modelling used the control variables spark advance, air/fuel ratio, and later exhaust gas recirculation, to optimize fuel consumption and emissions. On-line optimization of engines was a viable alternative to

developing these early models as engine testing for the control model took up to two weeks. Later, interest spread to modelling the catalyst by mapping the conversion efficiencies of CO, Hc, and NO_x to exhaust temperature, exhaust gas mass flow rate, and air/fuel (A/F) ratio.

A dynamic power-train model using eight variables (three relating to engine), and two time delays to model the effects of a four-stroke SI engine, automatic transmission, and rubber tires was reported [Cho 89]. The three engine variables considered are mass of air in intake manifold, engine speed, and fueling lag. The corresponding transport delays are the intake to torque production delay, and spark-to-torque production delay, both of which vary with engine speed.

Cho's work borrowed from a compact, nonlinear model for real-time control including intake manifold dynamics, fuel dynamics, process delays [Moskwa 87]. Although engine controllers were governing idle speed, A/F ratio, spark advance, and limiting knock, torque control to manage torque production and delivery was required in preparation for a complete power-train model for controlling shift quality and timing, and traction. Moskwa considered the engine as a group of five sub-systems (throttle body, intake manifold, fuel injection, combustion and torque production, and rotational dynamics).

Moskwa had added fuel injection and other refinements to one of the first physically based dynamic engine models that recognized throttle effects, intake manifold dynamics, and the discrete nature of the four stroke engine [Dobner 80,83]. Dobner's model's inputs were A/F ratio, throttle, spark advance, and load torque and provided net torque, and engine speed outputs. Dynamic aspects were handled by time delays and integration, and non-linearities allowed predictions over a broad operating range. The highly modular model divided the engine into carburetor, intake manifold, combustion and dynamics, with considerable time spent on fuel transport.

A nonlinear dynamic model including the effects of turbo-charging and inter-cooling (i.e., cooling the A/F mixture between the turbo-charger and the cylinder head) a SI engine has also been developed [Foss 89] to allow the evaluation of microprocessor control of the turbo-charger waste-gate.

2.3.3. Failure Detection and Isolation

Engine control models and detection filters can be used for real time diagnosis of sensor failures. Based on techniques from failure detection and isolation theory, a diagnostic system named the Binary Phase Detection Filter (BPDF) has been implemented [Min 89], [Rizzoni 89]. This system relies on functional redundancy between sensors to detect a wide range of sensor failures, and Rizzoni claims actuator failure detection is possible with similar methods. The BPDF detects and partially isolates failures, which is closer to identifying symptoms than to providing a diagnosis as a set of failed components. Considerable engine testing is required to obtain transformation matrices, and failures in other components of the electronic control system and engine can not be detected or diagnosed.

Willsky has reviewed design methods for failure detection in dynamic systems [Willsky 76] and concluded that the failure detection problem is extremely complex, and issues such as available computational facilities and level of hardware redundancy are critical. A more recent survey of Process Fault Detection based on modelling and estimation methods [Isermann 84] concluded that accurate models are critical, so only well defined and understood processes are suitable.

3. Automated Model-Based Diagnosis

This chapter describes our general approach to automated model-based diagnosis, providing a foundation for the modelling strategies presented later. The structure of the model and knowledge representation are presented in chapter 4, and implementation of the prototype model in Echidna is detailed in chapter 5.

Model-based diagnosis compares a model's predicted behaviour (values of variables, relationships, modes, states, conditions) with the actual behaviour exhibited by the artifact the model represents. Although the model is assumed to predict the model's component's behaviour correctly, it does not always predict healthy behaviour of the artifact. For example, if some of the model's components are in a faulty state, then the model-predicted behaviour will be faulty behaviour, but the model is assumed to correctly predict the (faulty) behaviour resulting from the faulty components. We will see later why the model is used for predicting both faulty and healthy behaviour. The assumption that the model correctly predicts the behaviour of the artifact is critical and questionable, and is treated in detail in chapter 4, but for now let us accept it.

Our automated diagnostic system is an interactive tool, not a substitute, for automotive technicians. Technicians should be able to direct the system, offer, reject and modify intermediate solutions and advice, and decide between equally promising paths. The system should be able to recover (and benefit) from the technician's mistakes. Many of the features necessary to achieve this (e.g. explanation systems) are not dealt with here, but where possible we work towards this goal.

The diagnostic procedure shown in Figure 3-1 is composed of three automated computational tasks: generating a set of symptoms, generating a set of

candidates ("candidate" will be defined, but for now think of candidates as possible explanations for the symptoms), and discriminating between those candidates; and one physical task, repairing or replacing a part of the engine or taking more measurements (and reporting back to the diagnostic system). An efficient computational implementation may combine portions of the first three tasks, but for clarity, and other reasons that will become clear later in this chapter, we will consider them separately. This procedure mimics that used by human diagnosticians, although humans are poor at remembering large amounts of data (long lists, for example), and so would necessarily use shorter lists and probably be more opportunistic in their approach. We will expand upon each of the three computational tasks after clarifying a few terms.

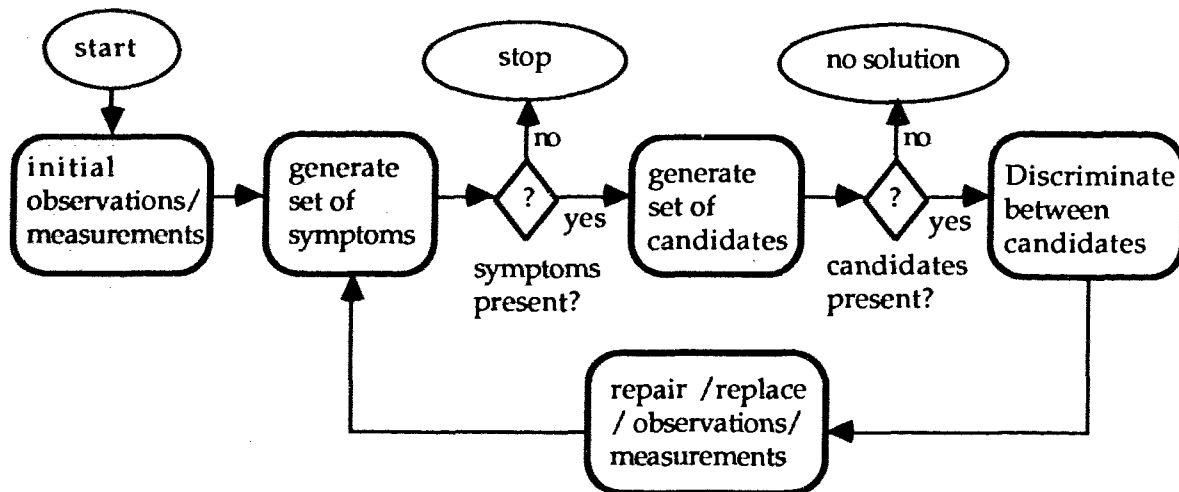


Figure 3-1
Diagnostic Procedure

The *artifact* (sometimes called the device) is the particular automobile engine that we are trying to diagnose (e.g., the engine in the red '88 Chevrolet Celebrity in bay number 1). From a diagnostic point of view it will be more interesting if the artifact is in a faulty condition, but we will not know its condition initially.

Each artifact is composed of an arbitrary number of *components*. We will discuss components in detail in chapter 4, but for now think of components as

the physical objects that can break and be repaired or replaced as a single unit. We explicitly represent both healthy and faulty component behaviour.

The *model* describes: 1) the artifact's internal structure, i.e., the interconnections between components, and 2) the behaviour of each component. The model we are using must have been developed for the class of engines that the artifact belongs to - in general, e.g., you can not use a model of a Volkswagen air-cooled 4 cylinder engine to diagnose a Cadillac V8 engine. The model may have a wide scope (e.g., covering all engines of a particular manufacturer, say General Motors), or it may be very specific and only cover a particular model, year, and range of serial numbers.

The model is sometimes referred to as a *deductive model* because it deduces the "output" variables of the artifact given its component's states (e.g., on, off, leaky, blocked, etc.), and known "inputs". We will argue in chapter 4 that this is an over simplification because the directionality of the model (i.e., which variables are inputs and which are outputs) may be indeterminate.

3.1. Symptom Generation

If the model is adjusted to represent a healthy car (i.e., all components of the model are working properly), and the predicted behaviour from the healthy car model disagrees with the actual behaviour of the artifact, then the artifact must not be working properly - it must have at least one faulty component, and it is displaying symptoms of that fault.

A *symptom* is a difference between the healthy car model's predicted behaviour and the artifact's actual behaviour (Figure 3-2). Symptoms can be qualitative, e.g., "the engine idles roughly", or quantitative, "the battery voltage is 8 volts and it should be greater than 9 volts". Note that quantitative symptoms can often be expressed qualitatively - e.g., "the battery voltage is

low". If an artifact exhibits no symptoms, then according to our model it is operating correctly, and no further diagnostic effort is required.

A faulty artifact will probably exhibit a number of symptoms, even if only one component is actually faulty. If the battery voltage is low, then the headlights will be dim, and the car will be hard to start. Multiple faulty components will only exacerbate this problem. If faults "mask" each other, it is possible that an artifact with faulty components exhibits no symptoms. Thus, an artifact with one or more faulty components will have zero or more symptoms.

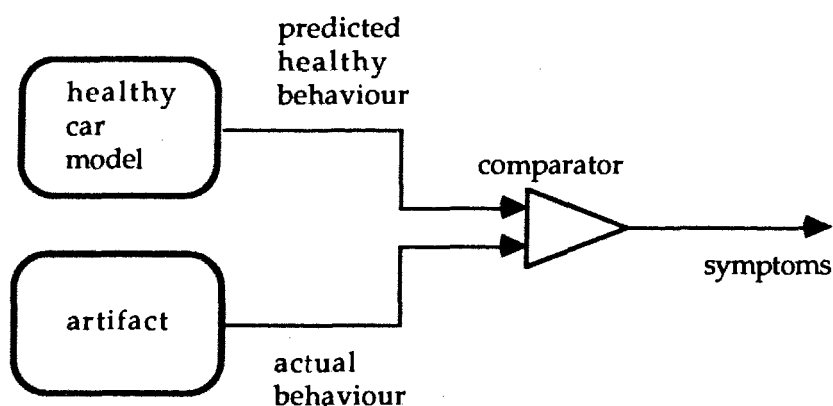


Figure 3-2
Symptom Generation

Our list of symptoms may grow or decline as the diagnostic session progresses and more data is obtained. New measurements may increase, or have no effect on, the number of symptoms. Replacing a faulty component may eliminate a portion of the symptoms, have no effect on the number of symptoms, or, in the case that that faulty component was masking symptoms, increase the number of symptoms. Note that symptoms are differences between the healthy car model's predictions and actual measurements, rather than differences between the healthy car model's predictions and potential measurements.

The diagnostic session begins by collecting information about the artifact. This information may include driveability symptoms supplied by the operator (e.g.,

the car is hard to start - the model predicts the car should be easy to start), observations made by the technician (e.g., the battery terminals are severely corroded - the model assumes good electrical contacts), and data available from the electronic control module (ECM) collected via a "scan tool" (e.g. the battery voltage sensed by the ECM). In general the operator of the vehicle will be unaware of symptoms generated directly from the ECM data. After the set of symptoms is determined we are ready to generate a set of candidates.

3.2. Candidate Set Generation

A *candidate* is an assignment of state (e.g., mode) to every component of the model, such that all model predictions are consistent with all evidence. There may be more than one candidate for each set of evidence. If no symptoms exist all components of a candidate will be in a healthy state. If one or more symptoms are present each candidate must include at least one component that is in a faulty state. As the diagnostic session progresses and more information is gathered, we want to converge on a single candidate that corresponds to the actual state of the artifact.

Commonly, diagnoses are described by the faulted components only, not as candidates explicitly describing the state of all components, both faulty and OK. We use the term *culprit* to describe the set of components that are in a faulty state. More than one candidate can have the same culprit.

If the model represents a faulty state, we refer to the differences between predicted and actual behaviour as *discrepancies*. By changing the state of chosen components in the model from good to faulty, (say, for example, by changing a particular wire's state from OK to open-circuited), we may be able to force the model's prediction of the artifact's behaviour to match the artifact's actual behaviour, eliminating all discrepancies. We have found a single candidate when there are no longer any discrepancies (Figure 3-3). We have

not yet said how to choose the components to implicate, but this will be answered shortly. The implicated set of components is the culprit, as defined above.

It is important to distinguish between symptoms and discrepancies. Symptoms can be thought of as special cases of discrepancies where the model is representing healthy operation. Symptoms are eliminated by fixing the artifact (car). Discrepancies are eliminated through modifying the model to match the artifact's behaviour.

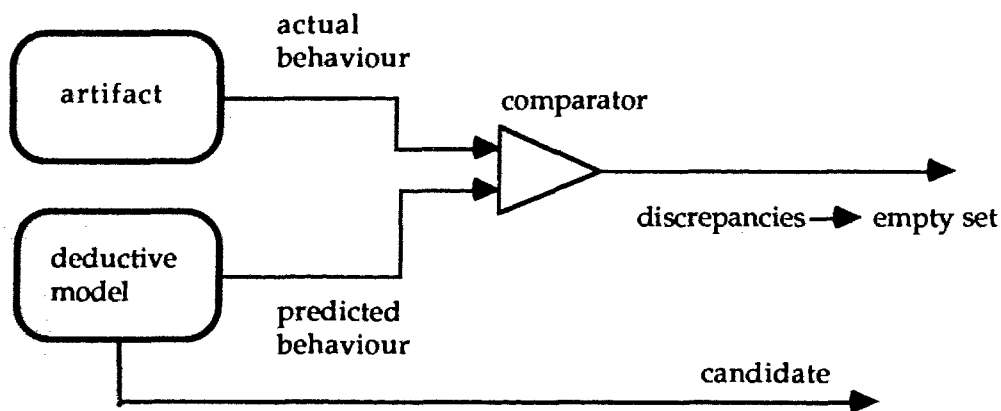


Figure 3-3
Candidate Generation

As noted earlier, there will probably be many candidates at the start of a diagnostic session. There may be so many that we do not care to generate them all. In this case we will have to decide how many to generate, either by some probabilistic measure or by deciding on some arbitrary maximum beforehand. We can generate more candidates later if new evidence forces us to rule out those already under consideration.

Some of the candidates may be very likely (e.g., weak battery), and some candidates will be quite unlikely (e.g., all the spark plug wires are shorted). It is preferable that the candidates be generated in order of likelihood for then we

are guaranteed that the most likely candidate will be in the set (it will be generated first). Ranking the candidates will later aid in discriminating between them. Generating candidates in ranked order will be discussed shortly.

To choose the implicated components we use search. Figure 1.1 (on page 6) shows an inference engine connected to the knowledge base (model). In traditional expert systems search is controlled by the inference engine. If the search strategy is sound (finds only valid candidates) and complete (finds all candidates) then it will find all valid candidates in the model. We must differentiate between the soundness and completeness of the search strategy and that of the diagnostic system as a whole. It is impossible for a model to cover all possible failures, and for efficiency, some popular search strategies are neither sound nor complete. Therefore there is no guarantee that we will be able to generate even a single candidate. If our model is unable to predict the actual behaviour we are measuring, then we will fail to find a candidate. We discuss the issue of completeness in detail in chapter 4.

We now present three methods of searching for candidates: naive search, abductive rule-based search, and probabilistic search.

3.2.1. Naive Search

Naive candidate generation uses the inference engine's built-in general search procedures to "blindly" search for candidates. It uses no domain-specific knowledge to guide the search and is generally diagnostically inefficient. Figure 3-3 corresponds to naive candidate generation if the inference engine (not shown in Figure 3-3) decides how to change the deductive model and monitors for the presence of discrepancies.

Depth-first search, as used by many popular inference engines, can not directly generate ranked candidates. The order in which such systems find candidates

depends on the syntactic ordering of the knowledge base. In effect, the system tries to find a logical proof for the top level goal, i.e., to find a candidate, or candidates, given known information. Ranking the candidates after they are generated in a non-ranked order offers no benefit in reducing search, as you have already committed resources to generate the candidate before you decide it is unlikely. The score of the candidate is calculated after it is generated, so equal time is spent on unlikely candidates. Unless all unranked candidates are generated there is no guarantee that the most likely will be included.

3.2.2. Abductive-Rule-Based Search

The disadvantage of naive candidate generation is speed. To increase speed, domain knowledge in an abductive rule-base (Figure 3-4) can heuristically direct the search for candidates. The diagnostic procedure described in this chapter distinguishes candidate generation from symptom generation precisely to allow for an abductive rule-base. We will describe the abductive rule-base in detail before tackling the possible pitfalls.

The abductive rules are symptom(s)/cause(s) pairs where the cause is a list (of arbitrary length) of faulty components which are hypothesized to cause the associated symptom(s). There is no restriction on the number of rules for each symptom or set of symptoms. Based on particular symptomatic information (and a strategy for choosing the best rule) the abductive rule-base will direct the inference engine to implicate (make faulty) certain components. If the implicated components cannot form part of a consistent candidate (i.e., the inference engine cannot assign healthy states to all other components such that all discrepancies are eliminated), the next rule is tried. If no rules match the symptoms, or all rules that match have been tried and failed, the abductive rule directed search fails.

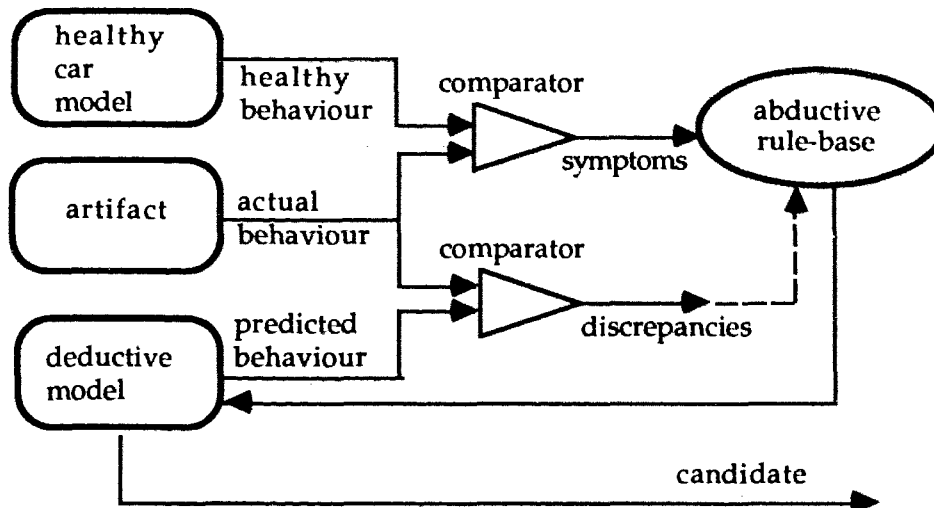


Figure 3-4
Abductive Candidate Generation

So far we have assumed that the abductive rule-base is symptom driven, and we know symptoms are eliminated only by replacing or repairing components of the artifact. There may be an advantage to allowing discrepancies to "fire" rules in the abductive rule-base. This corresponds to a "depth first" abductive search. For example the model's predictions computed from a symptom fired rule results in a number of discrepancies, and discrepancy fired rules modify the model to eliminate a fraction of these discrepancies, and so on. This loop continues until all discrepancies are eliminated, or no more rules match the discrepancies. Unfortunately, there is no guarantee that the number of discrepancies decrease after each rule fired by discrepancies. We have not yet explored the implications and efficiency of this strategy, but there are fundamental differences between discrepancy/cause pairs and symptom/cause pairs, and it is also not clear how or when this "feedback loop" will terminate.

One problem here is that it is not clear when to try a different symptom driven rule or another discrepancy driven rule. Discrepancy driven rules will add more faulty components, so will find candidates with many broken components before finding those with few broken components. Since the symptom/cause rules are heuristic rules, it would seem reasonable to try all rules matching the symptoms before resorting to the discrepancies.

A diagnostician might think "I believe component A is faulty, and I now have discrepancy X, which I think can be explained if component B is also faulty". The difference between this strategy and that presented above is that the technician took into consideration the faulty component A when he added faulty component B, rather than just basing his rule on discrepancy X. It seems that to be efficient, rules based on discrepancies must have as antecedents the faulty components that led to those discrepancies.

A serious drawback of the abductive rule-base directed search is in trading the inference engine, with its carefully planned and analyzed, though general, search strategy, for a rule-base complete with the weaknesses and strengths outlined in chapter 1.

The abductive rule-base is a global knowledge source. It implicates components based on symptoms, but the symptoms do not necessarily present themselves at the implicated components (e.g., the symptom "dim" may be manifested at the headlights and the implicated component may be an undercharged battery). The abductive rule-base, unlike the model, does not mimic the structure of the artifact and thus significantly reduces modularity of the knowledge base.

Time spent assembling and maintaining the abductive rule-base may be better spent on the model. A strong abductive rule-base may reduce the job of the model to that of verification, but an efficient model (and inference strategy) may eliminate the need for an abductive rule-base.

A record of the sections of the search space covered by the abductive rule-base should be maintained so if it fails to find enough candidates an alternate search strategy will not re-visit those sections already searched.

3.2.3. Probabilistic Search

We divide probabilistic search into two major sub-divisions, the first of which we call "global" strategies, and the second of which we will refer to as "local" strategies.

3.2.3.1. Global Probabilistic Strategy

As presented earlier we want to find the candidate with the highest posterior probability, i.e., $p(\text{candidate} | \text{evidence collected so far})$. (Note in chapter 2 we used "diagnosis" and here we use the more concise term "candidate" which was defined earlier in this chapter.)

Each component has a prior probability $p(\text{mode}_i)$ associated with each of its i modes:

$$\sum_i p(\text{mode}_i) = 1$$

Initially, before any evidence is gathered, and assuming independence of faults, $p(c_j)$ the probability of candidate c_j is [deKleer 89]:

$$p(c_j) = \prod_{m \in c_j} p(m)$$

where $p(m)$ is the prior probability of behaviour mode m being manifested (i.e., a particular component in a particular mode). After a variable x_i is measured to have a value v_{ik} :

$$p(c_j | x_i = v_{ik}) = p(x_i = v_{ik} | c_j) * p(c_j) / p(x_i = v_{ik})$$

where $p(x_i = v_{ik})$ is a normalization factor. Or in general, where e is the previously obtained evidence:

$$p(c_j | x_i = v_{ik}, e) = p(x_i = v_{ik} | c_j, e) * p(c_j, e) / p(x_i = v_{ik})$$

We want to generate the candidates in order of decreasing posterior probability (i.e., $p(c_j | x_i = v_{ik}, e)$), but we cannot calculate $p(x_i = v_{ik} | c_j, e)$ until after the candidate is generated. A solution is using a best first search based on an estimate of the probability of the candidate, that estimate being the value of $p(c_j | e)$ [deKleer 89]. The inference engine concentrates on the candidate with the highest estimated value.

This is a global probabilistic method because it uses the candidate probability, which is calculated from the probabilities of all components, to select the modes of individual components. It makes local decisions based on global information. The local strategy below attempts to reduce this dependence on global information.

3.2.3.2. Local Probabilistic Strategy

This strategy, often called a Bayesian Belief Network uses local summaries of global information [Pearl 88]. Although components still have access to global information, they only communicate with their neighbours. The Bayesian belief networks are fairly complex, so the following brief description omits many details.

Each component of the model shares variables with its neighbours, e.g., voltage, flow rate, displacement, etc. Each component receives probability distributions (i.e., the probability of each possible value) for those variables sent by its neighbours. In the case that the variable has a known value the probability distribution will be a spike at that value. Based on these distributions and the component's knowledge of the probabilities of its own state, it calculates a probability distribution for its state, chooses a most likely value for its state, and then notifies its neighbours of any changes. In this way

the "network" of components will choose states resulting in the most likely candidate.

For the remainder of this thesis we will concentrate on naive search because of its simplicity. The abductive and probabilistic methods may offer advantages, especially for efficient multiple fault candidates, but also cloud the issue of modelling. Both are heuristic, and therefore fallible, but may offer significant performance increases. We point out later some of the changes to our modelling strategy that would be necessary to incorporate these heuristic search strategies.

Once we have established a list of candidates we must determine which member of that list corresponds to the actual cause our artifact's symptoms. This is the process of discriminating between competing candidates.

3.3. Discriminating Between Candidates

The commercial use of an automated diagnostic system demands that it provide accurate diagnoses (i.e., culprits) more cheaply than an efficient (but computationally unaided) human diagnostician. Diagnostic costs (not including the eventual repair) include the use of space, equipment, and consumables, but the labour cost (i.e., technician's time) is assumed to be the dominating contributor. In some cases the repair time will be part of the diagnostic time (e.g., when a calibration fault occurs and is corrected during the diagnostic process). If the technician is standing by while computation is proceeding, the cost of the diagnostic session is proportional to the sum of the computational time and the technician's information-gathering and reporting time. The computational tasks mentioned above provide a list of candidates, and a possible further computational task is preparing an (optimum) strategy for gathering new information.

If we choose not to give the technician direction in discriminating between candidates the computational cost is eliminated, but discriminating between competing candidates and choosing measurements and tests becomes the sole responsibility of the technician. He may choose simplistic methods including blindly repairing or replacing some or all components implicated by likely candidates, or slightly better strategies based on component reliabilities (where this information is available). He may be able to develop a reasonable measurement and test strategy to test the candidates. These methods are labour-intensive, depend on the abilities of the technician, and will in general fail to minimize the dollar cost of the diagnostic session.

We assume that increasing computational effort on discrimination will reduce the demands on the technician and thereby reduce overall cost (which may not always be true, and depends strongly on computer technology). Ideally, we would like to be able to compute which series of measurements or tests would most cheaply lead to the correct candidate. Each measurement or test will eliminate a number of candidates, while confirming the possibility of others. The new information may or may not increase the number of symptoms but will decrease the number of candidates. This process can be likened to "divide and conquer", "differential diagnosis" or "half split". At this point, we need to distinguish between tests and measurements.

A *measurement* is the value of a particular variable while holding the inputs (and therefore outputs) constant. If suitable probabilistic information is known the entropy methods presented in chapter 2 can be used to estimate the next best measurement.

A *test* is a new set of artifact inputs (and possibly outputs) which will confirm some of the candidates and eliminate others. It is often easier to change (and measure) inputs and outputs than interior values. However, tests are much harder to generate, especially if any components have time dependence (i.e., their behaviour at time 2 depends on their behaviour at time 1). There does not seem to be a consensus on efficient methods for test generation.

Probabilities and likelihoods on their own are inadequate to provide minimal cost candidate discrimination. For instance, if the best next measurement (e.g., the measurement that will eliminate the highest number of candidates) is very expensive, and the second-best measurement is only marginally worse but very cheap, then we would like to make the second-best measurement first.

An ideal system would be more flexible than simply choosing between tests and measurements. There may be cases where it is cheaper to replace or repair components than perform distinguishing tests. Here, we would like to perform the repair and tell the diagnostic system that that component or set of components is definitely OK, and no candidate assigning them to a faulty mode is possible.

It is hoped that the technician will not introduce new faults, and therefore new symptoms, but, allowing for human and computational imperfection, all diagnostic sessions should end by confirming the lack of symptoms. Moreover, an ideal system would record the verified diagnosis in a database for future use, just as a human diagnostician would. This data-base would become an expert knowledge source and could be used for abductive rule-base generation, statistical updating, and automated learning. Automobile manufacturers could use this information to build more reliable vehicles.

4. Engine Modelling for Diagnosis

This chapter develops the knowledge representation we use for our prototype diagnostic model. We use simple examples within the automotive domain, but details of the prototype model are withheld until chapter 5. It is assumed the reader has read chapter 3, or is familiar with the concepts and definitions presented there. The syntax used in the code examples in this chapter is loosely based on that of the Echidna constraint reasoning system developed at Simon Fraser University's Center for Systems Science [Sidebottom 91]. Echidna's syntax is based on that of Prolog [Bratko 86], [Sterling 86]. Echidna code is presented in appendices A and B.

4.1. Completeness

One of the primary issues in diagnostic modelling, or modelling in general, is the level of abstraction of the model. Models at the lowest level of abstraction might deal with thermodynamic properties, chemical processes, atomic collisions, electrical charges, etc. Models at a higher level include those for ideal wires, ideal transistors, and ideal structural beams. Still higher are "black box" models of complete chips or circuit boards, or structural models used for the wind loads on buildings. Highest of all are concepts in a flow chart. A model at a low level of abstraction will be capable of calculating extremely precise answers, but may require massive computational resources. A higher level model will provide more general answers, presumably in less time.

Consider a model of a simple physical object at the lowest possible level of abstraction. Assume for a moment it is possible to encode everything we know about that object, including its particle physics, chemistry, etc. The model will be extremely large and complicated, and yet, at the edges of human knowledge and understanding, it will still be incomplete. So we cannot completely model

even a simple object [Davis 88]. Moreover, it is impractical, or maybe impossible, to encode everything that is known about even simple objects. As a consequence of these facts all knowledge bases will be incomplete at some level, and all reasoning systems using incomplete knowledge bases will be incomplete, however complete the inference strategy or problem solver.

Incomplete models can still provide much useful information. We have sent men to the moon with incomplete models of flight and interplanetary physics. Once we have accepted the fact that our model will necessarily be incomplete, and therefore incapable of finding all solutions, we must decide how detailed our model must be to be adequate for the chosen task.

The appropriate level of modelling abstraction depends on the context and use of the model. Take, for example, a thin metal rod formed into a coil. This coil can be modeled in many ways. If it is a heater element in a toaster its model will be concerned with electrical resistivity and thermal properties. If it is used to generate a magnetic field its model will include the physics of magnetic fields. If it is a spring in a mechanical spring/mass/damping system the important properties may be spring constant and maximum extension. Furthermore, a model adequate for the design of this coil may differ from models for other tasks, including diagnosis. It is possible a single model could be used for design and diagnosis, but each of these tasks may use only a portion of the model.

An automobile diagnostic system should identify failed components. The model does not need variables other than those that will achieve this goal. For instance, it is probably not necessary to calculate and report the changing thermodynamic properties of a failed electrical switch. However, it is desirable to be able to provide a description of how the failed component is behaving to convince the technician that the diagnosis is plausible. This would provide the starting point for an explanation system, which, as stated in chapter 1, will not form part of this thesis.

4.2. Competence

There is another measure that has been proposed to judge the acceptability of our model. A model of a physical component is said to be *competent* if, for each and every possible combination of the component's actual inputs and outputs, there exists at least one mode (this will be defined shortly, but for now think of a mode as a pre-defined relationship between input and output variables) whose predicted values are consistent with the actual values. (As mentioned earlier, it is at times unclear which variables are inputs and which variables are outputs. More on this later). Competence is certain if the component model has an unknown mode, i.e., a mode where the values of the inputs and outputs are unconstrained. This mode can cover any actual behaviour of the artifact - from OK to completely destroyed. The unknown mode provides little help in discriminating between likely candidates.

A model of an artifact is competent if, for each and every possible combination of the artifacts' inputs and outputs, modes can be selected for each of its components such that the predicted values of the artifact are consistent with actual values. This will be the case if all component models are competent, and the structure of the model (i.e., the way the components are connected) accurately represents the artifact. If a fault arises from a change in the artifact's structure from the design structure, say, for instance, the cases of two electrical sub-components touch and form a current path, then the structure of the model (which is based on the designed structure) no longer matches that of the artifact, and the two electrical component models will no longer accurately model the behaviour of their respective components. The model is not competent to explain changes in the artifact's structure. It may be possible to model changes in structure, but this type of modelling is highly context dependent (e.g., the physical location of a component may be critical). We have not explored this area.

4.3. Qualitative and Quantitative Models

Models at low levels of abstraction are often presented in a mathematical form. A detailed mathematical model based on sound physical principles calculates exact quantitative predictions. Developing such a model for a large system may be completely intractable, and even small artifacts may take excessive computer time. Our poor understanding of some systems (the human body, chemical processes, etc.) precludes the development of such models.

Human diagnosticians make use of qualitative reasoning. For example they may classify fuel pressure from the pump as acceptable, low, or high. The actual values are less important than the range they fall in. It is not necessary to calculate the exact pressure if it is known that it is too low. Whatever the result of the calculation, the pump will have to be replaced. Remember, the goal of automotive diagnosis is to identify a set of failed components, not to quantify the degradation of their performance

4.4. Assumptions and Limitations

Diagnosing large devices is a difficult task so simplifying assumptions are made to reduce complexity and increase efficiency. These assumptions further reduce the completeness of our model, but we have previously accepted that our model will necessarily be incomplete. Once again, the assumptions we make are acceptable if they allow us to develop a model that is adequate for our intended diagnostic task. Some of these assumptions may prove to be too unrealistic for a commercially viable system, while others may ultimately prove acceptable. We believe that the strategies presented here can later be modified in such a way as to remove any unacceptable assumptions.

4.4.1. Time - Steady State, Intermittent Faults

Many processes in an automobile are inherently oscillatory. Much of the literature on engine modelling uses the description "steady state" to describe an engine with constant inputs and outputs. Automobile inputs include, for example, the engine load (e.g. the grade of the hill being climbed), the throttle pedal position, steering, braking, shifting, the settings of all switches, and the operating environment (e.g. altitude). Artifact outputs are vehicle speed, acceleration, exhaust, noise, heat, air conditioning, etc. Internal engine variables are not restricted to single values under this definition of steady state. For example, while the engine is running the exhaust manifold pressure fluctuates throughout each engine cycle. However, at a given engine operating point (i.e., constant inputs) the shape of the pressure wave (i.e., peak pressure, average pressure, minimum pressure, the time between each of these events, etc.) will be constant. Even so, some variables will be decreasing during steady operation (the fuel in the tank is decreasing whenever the engine is running), and others will be increasing (the amount of fuel vapour stored in the charcoal canister), although the rate of change of fuel and vapour may be constant. This steady state definition effectively describes four distinct operating points: 1) key off, 2) key on engine off, 3) idle, 4) operation at a steady load and speed.

The above definition of steady state may prove to be necessary for the first commercially viable automotive diagnostic systems. Our prototype model has been developed with a more strict definition of steady state. At steady state all the artifact's inputs and outputs are constant, and all variables and component states are constant. This assumption allows us to ignore the effects of time. Components such as charcoal canisters and fuel tanks, as mentioned earlier, have variables that change even as the artifact inputs and outputs are held constant.

Our steady state assumption precludes the diagnosis of intermittent faults. Intermittent faults are those that cause symptoms to appear and disappear seemingly at random. The artifact exhibits no symptoms most of the time, but occasionally exhibits a particular set of symptoms. Symptoms can only be removed by replacing a faulty component. Unfortunately many electrical faults are of the intermittent variety.

We assume that all measurements made and reported by the technician are accurate and are taken at the correct location. When a variable is bound to a measured value, that value is assumed constant and correct for the duration of the diagnostic session. The measurements may not all be taken at the same time, but all variables are assumed to remain constant.

4.4.2. Single Fault Assumption

If we consider all multiple fault candidates (i.e., more than one component can be in a faulty state) then the number of possible candidates explodes exponentially. An artifact with N primitive components with an average of K behavioural modes will have on the order of K raised to the power N potential multiple fault candidates. An artifact with 10 components each with four behavioural modes will have on the order of 4 raised to the power 10, or approximately 1,000,000 candidates! Limiting the maximum number of faults allowed in each candidate reduces the size of the search space, but also eliminates potential, although unlikely, candidates (i.e., those with more than the maximum number of faults).

The single fault assumption allows candidates implicating a single component, but is unrealistic in many situations, as will be discussed shortly. If we limit ourselves to single fault candidates, and disregard additional candidates with the same implicated component and state, then we have potentially as many candidates as the sum of the faulty modes for each and every primitive

component. An artifact with N components with an average of K behavioural modes, F of which are faulty ($F < K$), will have $F \cdot N$ candidates. For the example given above, if three of each component's modes were faulty, there would be $3 \cdot 10$, or 30 single fault candidates.

Some sets of symptoms will not be explainable by a single fault candidate and if we limit ourselves to this assumption we may be incapable of finding the correct candidate, or even a single candidate. The model is no longer generally competent, because it will not be able to match the behaviour of an artifact with more than one faulty component. The model could still be competent for single fault candidates. Having an unknown mode would accomplish this. However, the unknown mode is more unlikely than the explicitly represented modes, but will be responsible for many candidates. We have decided to exclude the unknown mode, assuming the explicitly represented modes to be much more likely. Our model is neither complete nor competent.

In general an artifact must have a single fault before it has multiple faults, unless the multiple faults occur simultaneously. Multiple faults that occur simultaneously are probably not independent. If healthy component operation is much more likely than faulty operation, and all faults are independent, then usually single fault candidates will be more likely than multiple fault candidates. However, modern automotive electronic controllers can compensate for some minor faults, so the operator of the vehicle is unlikely to experience symptoms until multiple faults exist. The single fault assumption will often fail in this situation.

The single fault assumption may be useful for preventative maintenance (i.e., the operator has not yet experienced symptoms, but the technician finds faulty data coming from the ECM). Here, the number and significance of failures must be small. A search for single fault candidates that did not find enough candidates, or did not find the correct candidate, could be followed by searches for double fault candidates, then triple fault candidates, etc.

4.4.3. Correct configuration of components

It is assumed that the model structure accurately reflects that of the artifact. The model cannot re-arrange its component models, so cannot predict the output of an improperly connected artifact.

4.4.4. Measurements and tolerances

The issue of measurement tolerances is not considered. For example, if a variable is predicted to have a value of 6.0 volts, but its actual measured value is 5.75 volts, is the measured value consistent or inconsistent with the predicted value? Possible decision schemes include fixed tolerances (say 0.5 volts), fixed percentages (say 10%), or, better yet, tolerances that are local to the variable being measured. We have assumed all variables have discrete integer values and the tolerance is one unit. Although unrealistic, this assumption is adequate for our model's level of abstraction. For a full commercial implementation a more complex scheme would merit consideration.

4.5. Component Models

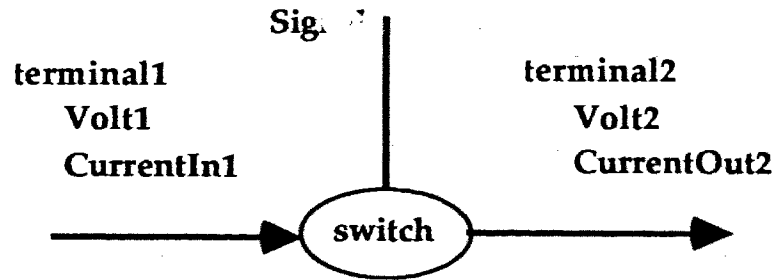
There are two basic types of component models, primitive component models and composite component models. In section 4.5.1. we will define and discuss primitive component models, providing a foundation for the discussion of composite component models in section 4.5.2.

4.5.1. Primitive Component Models

Primitive component models represent fallible physical components that are replaced or repaired as single units. The spark plug wire for the number 1 cylinder is an example of a physical component that could be represented by a primitive component model.

Each primitive component model has: 1) an arbitrary number (greater than 0) of interface variables which it can communicate to the outside world, 2) a finite and exhaustive set of behavioural modes relating those interface variables.

Interface variables may be qualitative or quantitative, discrete or continuous. Our example `simpleSwitch` (Figure 4-1) has five interface variables - `Signal`, `Volt1`, `Volt2`, `CurrentIn1`, and `CurrentOut2`. (See [Sidebottom 91] for details of Echidna syntax, and appendix A.2. for a detailed code for a switch) `Signal` is a boolean (qualitative) variable (it is *off* or *on*), and the others are numerical (quantitative). Interface variables can have a unique value, or they can have a defined range of values. For example, `Volt1` could have a single value of 12 volts, or a range of 9 to 16 volts.



```

simpleSwitch
{
% list of interface variables
Signal.
Volt1.      CurrentIn1.
Volt2.      CurrentOut2.

% groupings of variables to terminals
terminal1(Volt1,CurrentIn1).
terminal2(Volt2,CurrentOut2).

% behavioural modes of simple switch
mode :- offMode; onMode; shortedMode; openCctMode.

offMode :- Signal = off, CurrentIn1 = 0, CurrentOut2 = 0,
           State = off,
           Condition = good.

onMode :- Signal = on, CurrentIn1 = CurrentOut2, Volt1 = Volt2,
          State = on,
          Condition = good.

shortedMode :- CurrentIn1 = CurrentOut2, Volt1 = Volt2,
              State = shorted,
              Condition = bad.

openCctMode :- CurrentIn1 = 0, CurrentOut2 = 0,
               State = openCct,
               Condition = bad.
}

```

Figure 4-1
Primitive Component Model- Simple Switch

We commonly think of physical components as having input variables which cause output variables. This is not generally the case, however. Take, for instance, our simple switch with two electrical connections, each of which can be represented as two variables, one for voltage (relative to ground) and one for current. We all might agree that Signal is an input, but either electrical connection could be the input, and the other would be the output. We think of interface variables coexisting with each other rather than causing each other. Some interface variables require a defined direction (e.g., current), so where necessary a positive direction will be arbitrarily assigned. Otherwise, directionality is insignificant. In our example CurrentIn1 is defined for positive flow in, and CurrentOut2 for positive current flow out.

Behavioural modes, also known as states, or simply modes, are consistent sets of relationships between the values of the interface variables. Our example simpleSwitch has four behavioural modes - offMode, onMode, shortedMode, and openCctMode. Behavioural modes can represent healthy operation (i.e., a switch can be on or off), or faulty operation (a switch can also be shorted or stuck open). Behavioural modes are not required to define exhaustively the relationships between all interface variables. For example, when a simpleSwitch is off, its voltages are unknown and offMode does not limit, or even mention, them.

Where two or more variables are intuitively related, as in an electrical connection, it is convenient, although not necessary, to group them into a single entity we call a terminal. Our example has two terminals, named terminal1 and terminal2, both representing electrical connections.

As stated previously, a primitive component model may have several modes that represent healthy (or good, or OK, or correct) operation, and several modes that are faulty (or bad, or notOK). We define *Condition* to take the value of "good" or "bad". Our switch has two modes with Condition = good (offMode and onMode), and two modes with Condition = bad (shortedMode and openCctMode). Note that a switch that is stuck open behaves exactly as a switch

that is turned off, so as long as that switch remains off, either `offMode` or `openMode` accurately describes its behaviour. `OffMode` should be more likely than `openMode` so it should be chosen first. Our model relies on the ordering of the modes to achieve this.

We use a black box analogy for primitive component models. A primitive component model is a single black box. Interface variables pass through holes on the outside of the box. Each black box has a rotary knob on the outside to select the behavioural mode. The knob positions that correspond to a healthy mode cause a green light to shine, and the positions corresponding to a faulty mode will cause a red light to shine. From the outside of the box the internal structure of the component can not be seen.

Our model of a simple switch does not refer to a particular switch, in a particular setting. It is tempting to think we have a general, and context independent description of how a simple switch behaves. It is, however, a mistake to believe that any component can be competently modelled without respect to the context in which it operates. A heavy switch may be used as a door stop, but our model would not be adequate to predict its behaviour for this task. Our model of the switch is useful only when it represents an electrical control device in a simple electrical circuit. In this limited context the switch model is general, and the actual values of the interface variables depend on the components connected to the switch. The generality of the model, within its limited context, allows a number of desirable features including modularity, code reuse, and hierarchical inheritance, which will be discussed later.

4.5.2. Composite Component Models

We could represent an engine as an extremely large set of interconnected primitive component models. We notice, however, that certain groups of components work together closely to perform a required task. Our cognitive

load is reduced by reasoning about a group of cooperating components before directly implicating a specific component. For example, certain symptoms may lead to the hypothesis that a fault lies within a group of components. After hypothesizing that that group is mis-behaving, we would search for the member(s) of that group that are responsible. Instead of immediately dealing with a large number of objects, we are considering a much smaller number of groups, and each group we eliminate vastly reduces our search. When we do this we are implicitly changing the level of abstraction of our mental model of the engine. These groupings of physical components also provide a mental organization for complicated artifacts.

Composite component models represent groups of two or more sub-components that work closely together to perform a desired task. A *sub-component* may be represented by either a primitive component model or another composite component model.

We have a black box analogy for composite component models. A composite component model is a single, larger black box. The composite box has a number of holes through which its interface variables can pass. Each sub-component is a black box inside the composite box, but the sub-component boxes can not be seen from the outside. Some of the interface variable holes in the sub-component boxes will align with those in the composite box. Some of the interface variable holes in the sub-component boxes will align with those of other sub-component boxes. All of the knobs for selecting the state of the sub-components are accessible from the outside of the composite box, and all of the sub-component condition lights are visible from the outside of the composite box.

There are two categories of composite components, compound components and system components. In section 4.5.2.1. we will first describe the system type of composite component, followed by a discussion of the compound component in section 4.5.2.2.

4.5.2.1. System Component Models

A fuel injection system could be represented as a system component. The sub-components of a fuel system work together to deliver clean fuel, at the correct flow rate and pressure, from the tank to the injectors. The sub-components include hoses, injectors, a pump, a filter, a tank, and a pressure relief valve. A fault in any of the sub-components may adversely affect the flow of fuel to the cylinders.

How do we decide which components to include in a given system? Some components are easily excluded from a system, e.g., tires would not likely be included in the fuel injection system. Other components are closely related, but intuitively are not part of a system. The battery provides 12-volt power to the fuel injectors, but most people would not consider the battery part of the fuel system. A faulty battery would affect the delivery of fuel, but would also have many other far reaching effects, some of which may be more noticeable, such as the refusal of the starter to turn over.

It can be more difficult to determine intuitively whether a particular component belongs in the system. The fuel pump draws 12-volt power from a fuse. Is that fuse part of the fuel system, or part of the 12 volt power system? We suggest it belongs in the fuel system if the fuel system is the only system it protects, but would include it in the 12-volt power system if it protected other systems as well. These grouping decisions may be somewhat arbitrary, but a reasonable solution is possible because the engine designer and fuel system designer probably traveled a similar path, and choosing different boundaries should not strongly influence the diagnostic routine.

A simple example of a system component is the imaginary twoSwitches in Figure 4-2. It is composed of two sub-components named S1 and S2, each of which is a simple switch behaving as described in the earlier example of

primitive components. Just like primitive component models, system models have interface variables and terminals. System twoSwitches has six interface variables, SignalA, SignalB, VoltA, CurrentInA, VoltB, CurrentOutB, and two terminals, terminalA, and terminalB. The directionality of interface variables is as described for primitive component models. Component S1 has its terminal1 connected to twoSwitches terminalA, and its terminal2 is connected to S2's terminal1. The other interface variables of S1, S2, and twoSwitches are connected as shown.

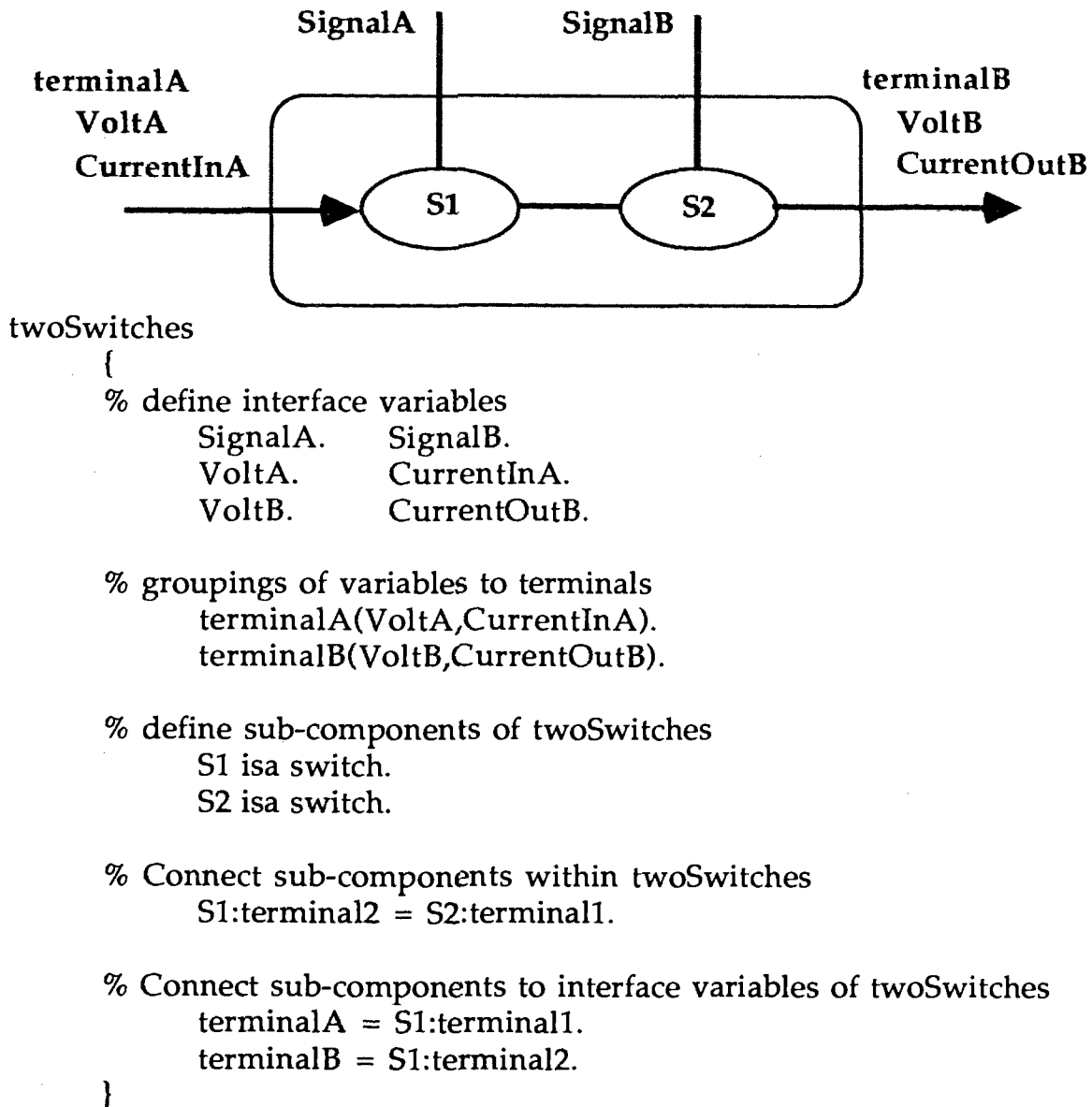


Figure 4-2

Composite System Component - twoSwitches diagram

Composite components (i.e., both system and compound components) differ from primitive components in that their behaviour is not explicitly represented, rather their behaviour is the sum of the behaviour of their sub-components. The behavioural mode of a composite component is a relation over the behavioural modes of its sub-components. This relation could take a number of forms. It could be a mapping into a single variable, with each

possible value representing a unique combination of the modes of the sub-components. We use the vector (or list) of the states of the sub-components as the state of a system component.

System components are not replaced as a unit. A faulty system results in the replacement or repair of one or more of its sub-components. System components can not in themselves fail, but are considered faulty if any of their sub-components are faulty.

The condition of a system component is a relation over the condition of its sub-components. The simplest such relation is that the system condition is good if all of its sub-components is good, and bad if any of its sub-components is bad. A slightly more insightful scheme is labeling the condition of a faulty system with the name of the failed sub-component i.e., the condition of a faulty fuel system could be `bad_injector`. We use a similar approach to that used above for modes, where the condition of a system is a vector of the conditions of its sub-components. From this vector we can determine whether any sub-components have failed, and which components have failed.

The previous discussion of context of primitive components holds for system components also. The `twoSwitches` example is a general model of a two switch system only in the context of a simple electrical circuit. In this context `VoltB` and `CurrentOutB` depend on `VoltA`, `CurrentInA`, `SignalA` and `SignalB`, as well as the behavioural mode associated with each switch.

4.5.2.2. Compound Component Models

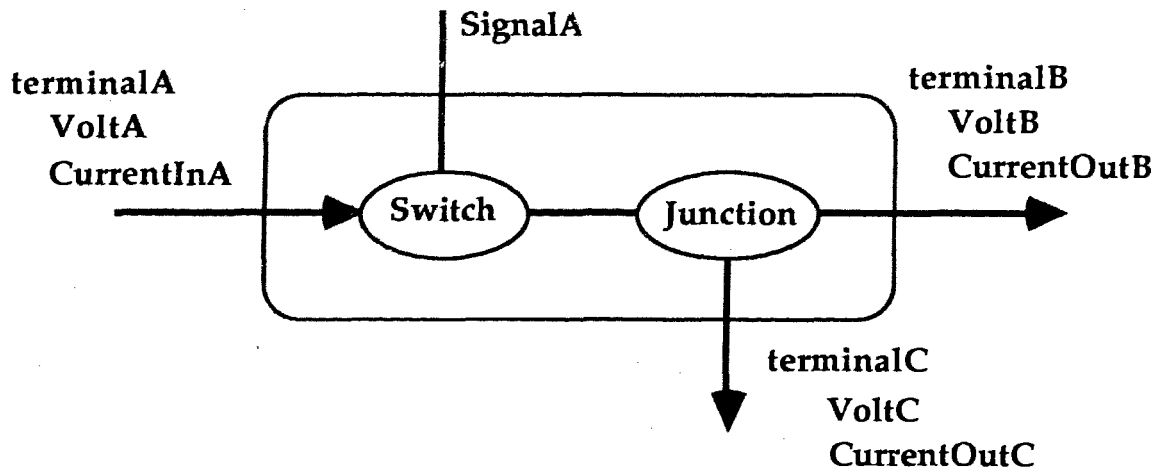
A compound component model represents a single complex physical component as a set of simpler sub-component models. The fundamental difference between a compound component model and a system component model is that the compound model represents a component that will be

replaced as a single unit. Any object that can be represented with a compound component model could also be represented by a primitive component model.

An ignition switch can be represented as a compound component. An ignition switch is a complex device with many poles and throws, and defining behavioural modes would be tedious. Alternately, the ignition switch can be modelled as a collection of linked simple switches and junctions, where each switch acts as a primitive switch and each junction acts like a primitive junction.

Like system component models, the condition of a compound component model is a relation over the condition of its sub-components. However, the compound component model represents a single physical component. The sub-components can not be replaced independently. We therefore summarize the conditions of the sub-components into a scalar condition for the compound component. If any of the sub-components is faulty (i.e., in a bad condition) then the condition of the compound is bad. Otherwise, the condition of the compound component is good.

A simple example of a compound component model is the imaginary ignition switch in Figure 4-3. It is composed of two sub-components, one named Switch which behaves like a simple switch, and another named Junction, which behaves like a primitive component model for a junction, (i.e., a connection between two or more conductors). We have not given a detailed description of a junction, but it would have a form similar to that of our simple switch with a good mode and faulty modes. Just like primitive and system models, compound component models have interface variables and terminals. Compound ignitionSwitch has seven interface variables including SignalA, and three terminals including terminalA. The directionality of interface variables is as described for primitive and system models. The other interface variables of Switch, Junction, and ignitionSwitch are connected as shown.



```

ignitionSwitch
{% define interface variables
    SignalA.
    VoltA.    CurrentInA.
    VoltB.    CurrentOutB.
    VoltC.    CurrentOutC.

% groupings of variables to terminals
terminalA(VoltA,CurrentInA).
terminalB(VoltB,CurrentOutB).
terminalC(VoltC,CurrentOutC).

% define sub-components of ignition switch
    Switch isa switch.
    Junction isa junction.

% Connect sub-components within ignition switch
    Switch:terminal2 = Junction:terminal1.

% Connect sub-components to interface variables
terminalA(VoltA,CurrentInA) = Switch:terminal1(Volt1,CurrentIn1).
terminalB(VoltB,CurrentInB) = Switch:terminal2(Volt2, -CurrentIn2).
terminalC(VoltC,CurrentInC) = Switch:terminal3(Volt3, -CurrentIn3).

% define behavioural modes of ignitionSwitch
    mode:- goodMode; badMode.
    goodMode :- Switch:Condition(Good),
                Junction:Condition(Good),
                Condition = Good.
    badMode :- Switch:Condition(Bad) or Junction:Condition(Bad),
                Condition = Bad.
}

```

Figure 4-3

Composite Compound Component - ignitionSwitch diagram

Like system component models, the behavioural mode of a compound component model is a relation over the behavioural modes of its sub-components, and we use the vector (or list) of the behavioural modes of the sub-components as the state of a compound component.

The context dependent generality of compound component models is as described for primitive and system models.

4.6. Hierarchical Composition

The use of primitive and composite components leads directly to a hierarchical structuring of the model. Two separate types of hierarchies result: a specialization hierarchy and a composition hierarchy [Genesereth 81]. Both types of hierarchies promote modularity, code re-use, and representation of knowledge at varying levels of abstraction.

4.6.1. Specialization Hierarchy

Specialization hierarchies, also known as is-a, kind-of, and type hierarchies [Luger 89], are tree shaped structures that allow objects (in our case components) to inherit from other, more general objects. In a specialization hierarchy, each node is an instance of its parent, but a more detailed or "specialized" instance.

Figure 4-4 shows the top levels of our specialization hierarchy. Echidna code for the top of this hierarchy (component, primitive component, compound component, and system component) is shown in appendix A.1. Our root node is a general component model, and attributes (i.e., the existence of interface

variables, state, and condition) that are common to both primitive component models and composite component models are stored here.

All primitive component models inherit the attributes of the component model and add a specific structure including behavioural modes. A switch is a specific instance of primitive component model. Although not shown here, our simple switch may have children that represent particular manufacturer's models of switches, which may have limited current capacity and voltage, and may have additional behavioural modes.

Composite components inherit the attributes of components, and all have sub-components. Compound components inherit the attributes of composites, and add scalar "Condition" (e.g., while "component" declares that all components have "Condition", "compound component" declares that the "Condition" of compound components will have a single value). System components have the attributes of composites but have a vector representation for condition. If this hierarchy covered a family of engines it might have several children inheriting from a general fuel system.

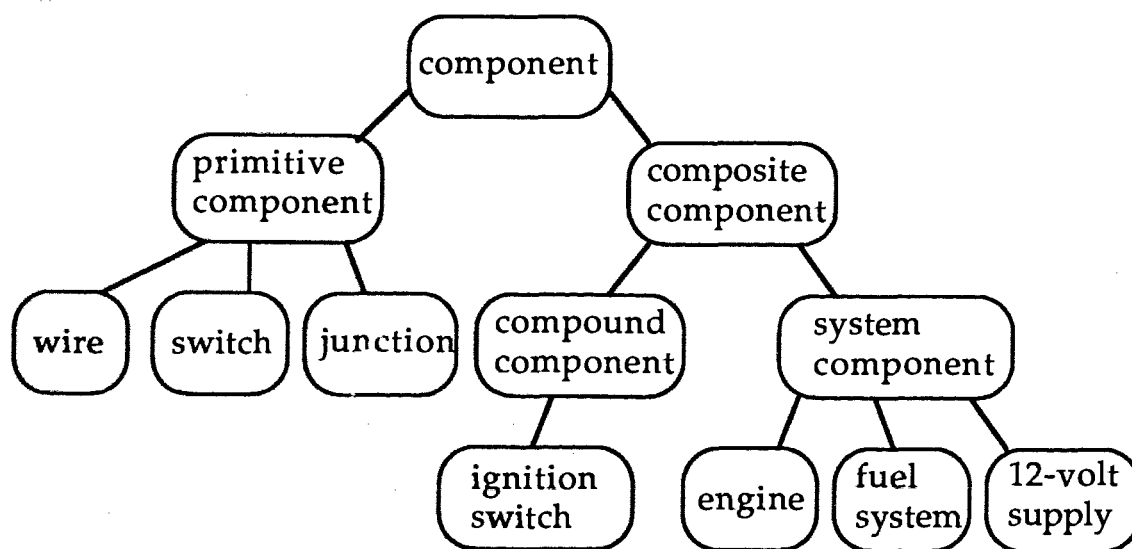


Figure 4-4
Partial Vehicle Specialization Hierarchy

4.6.2. Composition Hierarchies

The use of composite system components leads directly to a composition, or part-of, hierarchy, a tree structure where the nodes are components and the arcs are part-of links. Each parent node is "composed" of its child nodes, or in other words, the sum of a parent's children is that parent itself. The root node is a system model of the complete artifact and the leaf nodes are primitive and compound component models. For example, Figure 4-5 shows a vehicle (i.e., a composite system component), with sub-components including engine and electrical, each of which is also a composite component, and has sub-components of its own (some of which are shown). Primitive and compound components are not shown in Figure 4-5, but would be at the far right.

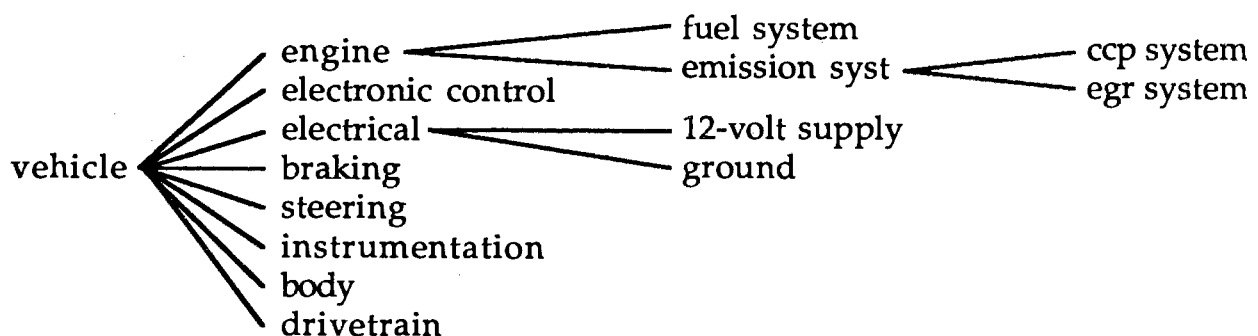


Figure 4-5

Partial Vehicle Composition Hierarchy

It is important to understand that objects in a composition hierarchy may be connected even though the composition hierarchy shows no links between them. The tires are certainly not part of the engine, but they are definitely connected or the vehicle could not move.

The beauty of this structure is that all parents of a component are affected by the state of that component, and all other components are unaffected by its state. For example, a broken fuel injector leads to a faulty engine, and in turn, a

faulty vehicle, but does not lead to a faulty electrical system. A faulty battery leads to a faulty 12-volt supply, and electrical system, but does not lead to a faulty engine. Since all components are descendents of the root node (vehicle), any fault will lead to a faulty vehicle, as we expect.

4.7. Suggestions for Future Work on Modelling

The assumptions listed earlier in this chapter may be unrealistic - in particular the single fault assumption, discrete valued variables, and the unit tolerance. Modelling structure and configuration (i.e., having modes for system components as well as for primitive components) may allow diagnoses implicating a change in structure. Our modelling strategy does not yet support three important areas: ranking candidates, discrimination between candidates, and time considerations.

4.7.1. Ranking Candidates

The modelling strategy presented here does not provide information required for candidates to be generated in ranked order. As discussed in chapter 3 probability theory, and possibly abductive rules, may have the potential to achieve this goal. Probability theory, whether global or local in strategy, needs the explicit representation of the prior probabilities associated with each behavioural mode. If we store these probabilities with our "general" models described above we are saying "all physical components that behave in the fashion our primitive model describes, have these failure probabilities". But we know, for instance, although all wires behave similarly, their failure probability depends on the conditions they operate under and the standards to which they were manufactured and installed. We conclude that all probabilities are context dependent, and would vary between instances of a single model. The power of the probabilistic methods lies in having the

probabilities associated with the instances of the models corresponding to particular physical components.

4.7.2. Discrimination between Candidates

As mentioned earlier we want to eliminate candidates until we have all but the one that corresponds to the actual state of the artifact. We want to minimize the cost of eliminating candidates. The entropy calculations described in chapter two can be used for this but are based on normalized probabilities. A method minimizing the expected dollar cost of the diagnostic session would be better yet. Abductive rules may also be able to help here, but we did not have the resources to explore these options.

4.7.2. Time

Some method of dealing with time is required. Some components, such as fuel tanks and charcoal canister can not be effectively modelled without reference to time. For these components the outputs depend on past inputs, i.e., there is a time delay between values of interface variables.

5. Prototype Diagnostic Model

This chapter presents the prototype automobile diagnostic model, preceded by a quick description of the Echidna constraint reasoning system. The performance of the Echidna implemented model, and obvious improvements to that implementation, are discussed. Echidna code, input files, and output data is shown in the appendices.

5.1. Echidna Constraint Reasoning System

The prototype model presented in this chapter was implemented using the Echidna constraint reasoning system developed at Simon Fraser's Center for Systems Science [Sidebottom 91], and running on a Sun SPARCstation 1. Echidna was developed for synthesis, analysis and other recognition tasks, and couples constraint satisfaction, logic programming based on the first-order Horn-clause semantics of Prolog, justification-based truth maintenance, and dependency backtracking, all in an object-oriented framework. In Echidna objects are represented as schemata with persistent state. Objects are accessed by unifying goals (logical messages) with the predicates (logical methods) defined within the schema.

Echidna was being developed concurrently with our prototype model, and was still at an early stage of development, so many advanced features were not yet available. The performance of our diagnostic system depended on our modelling efficiency and the reasoning efficiency of Echidna. Recent advances to Echidna have provided a richer set of tools for reasoning and knowledge representation, and would probably prove to be much more efficient.

The early version of Echidna was unable to reason about, or even remember, multiple solutions (i.e., multiple candidates). To find additional candidates the

fail predicate must be used, which eliminates the present candidate. Before issuing the fail goal the present candidate can be displayed on the screen or stored in an external file, but no more reasoning about that candidate can take place. To work around this problem we computed a completely new set of candidates each time new data were added. This was clearly inefficient, as time was spent reproducing previously computed results. This problem precluded the implementation of a minimum entropy calculation to discriminate between candidates. This type of calculation must have all candidates available simultaneously.

If the multiple solution problem were eliminated and the entropy calculation performed, the Echidna based diagnostic system could tell the technician which measurement to perform next. The technician might like to respond with the value resulting from that measurement, but the Echidna interface required that the technician enter that measurement as a top level Echidna goal. Echidna could not monitor the terminal for data input, it could only respond to top level goals typed in the Echidna window. The graphic interface for Echidna is being developed at this time.

5.2. Prototype Vehicle

The prototype vehicle used as a guideline is a 1988 Chevrolet Celebrity with 3.8 liter v-6 engine, vehicle identification number (VIN) W. The VIN W engine has an advanced electronic control system, and has undergone considerable analysis and testing. Service and engineering documentation, and technician's expertise, is available.

The vehicle was decomposed into major sub-systems. (Remember, sub-systems are actually composite system components as described in chapter 4). The major sub-systems were: engine, drive-train, body, electrical, electronic control, instrumentation, steering, braking, climate control. This decomposition was

intuitive (at least to the author), but, as stated earlier, somewhat arbitrary, and probably other groupings could serve equally well. The most controversial decision is separating the electronic control system from the engine, but the electronic control system governs many components not directly related to the engine - for instance, components that regulate the temperature in the operator compartment. No pretense is made that these groupings are adequate and sufficient for commercial implementation, they are presented only as an example.

We did not have the resources or the desire to model a complete engine or family of engines, we decided to concentrate on a subset of the major sub-systems rather than all components connected to the electronic control module (ECM). The principles developed should be widely applicable.

5.3. Prototype Sub-systems

It was tempting to explore the electronic control system itself, but there is a critical shortage of detailed engineering information about the control system because of the automobile manufacturers' reluctance to release these data. The techniques developed should be equally applicable to components of the electronic control system, when details of their design become available.

We decided to concentrate on a portion of the 12 - volt power system (for ease of model development and explanation), the charcoal canister purge (CCP) system, (because it incorporates mechanical components, and emission control systems including the CCP system are relatively unreliable), and the fuel system (because it is the system most influenced by the electronic control system). The prototype model composition hierarchy is shown in figure 5-1. All of the boxes shown in this hierarchy are represented as composite system components. Echidna code for systems is shown in appendices A.1. and A.4. .

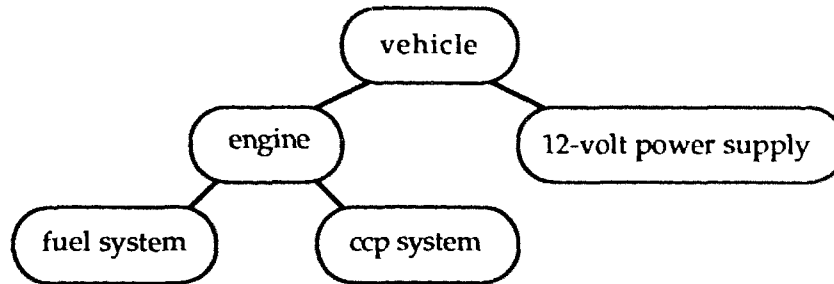


Figure 5-1
 Prototype Model Composition Hierarchy

The 12-volt power system distributes 12 volt DC electrical power from the battery to all systems that have low voltage (i.e., 12 volt) electrical components. Our model of a portion of the 12-volt power system is shown in Figure 5-2. A complete model would have many more sub-components and many more interface variables. The sub-components of this system model are three wires, two fuses, two junctions, an ignition switch, and a battery. The ignition switch is represented by a compound component model, and all other sub-components are represented by primitive component models. Echidna code for primitive component models is shown in appendices A.1. and A.2., code for compound components is shown in appendices A.1. and A.3., and code for the power system is presented in appendices A.1. and A.4. The battery is unrealistically modelled with only one terminal, but the model is adequate for the intended use. The 12-volt power system shares interface variables, as defined in the figure, with the operator, exhaust gas recirculation (EGR) system, CCP system, fuel system, and electronic control system. We did not model the EGR system.

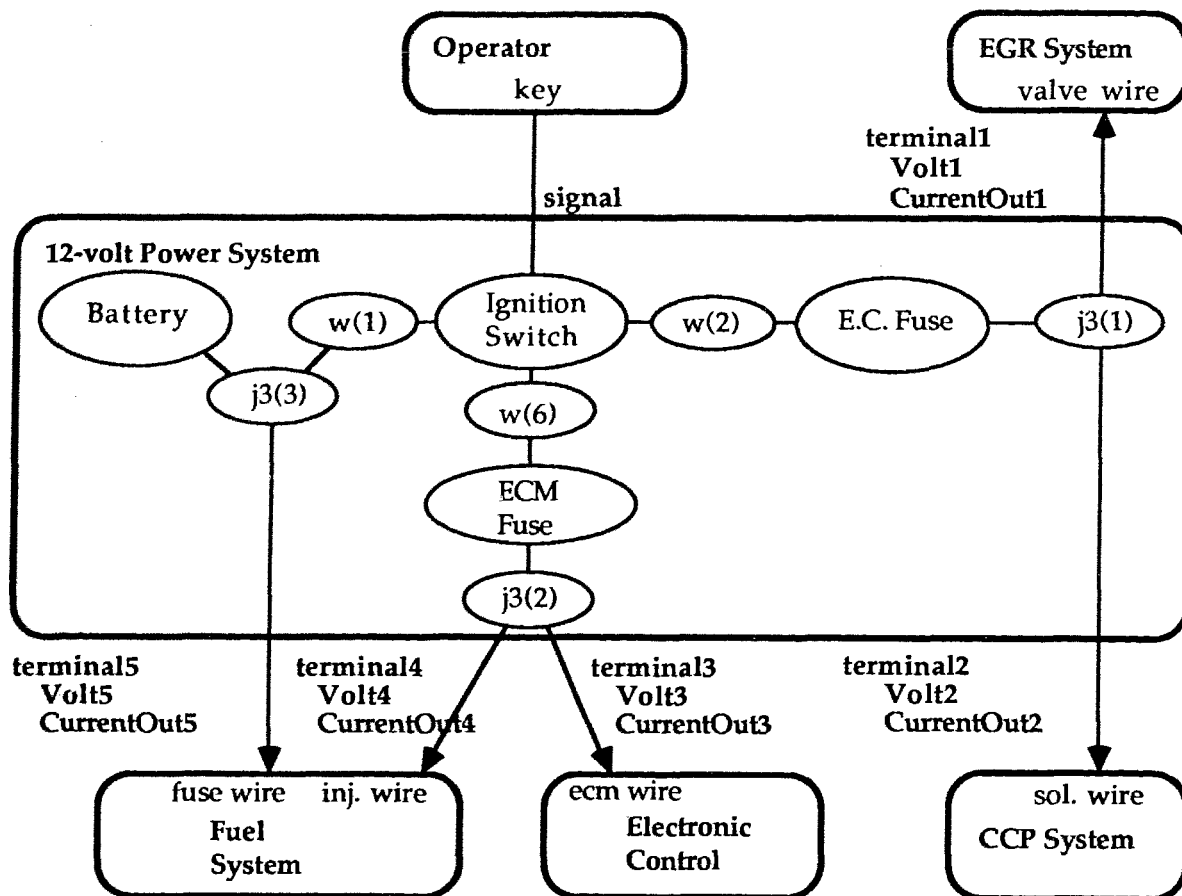


Figure 5-2
12-Volt Power Distribution System

The CCP system (Figure 5-3) eliminates fuel evaporation from the fuel system to the environment by storing the vapour in a charcoal canister, and venting that vapour to the manifold when commanded by the electronic control system. The sub-components of the CCP system are five hoses, two wires, a purge solenoid, charcoal canister, and a pressure control valve. All sub-components are represented as primitive component models. The system shares interface variables with the 12-volt power system, air system, electronic control system, atmosphere, and fuel system. The interface variables are defined in the figure. We did not model the air system. Echidna code for the primitive sub-component models are found in appendices A.1. and A.2., and code for the CCP system is presented in appendices A.1. and A.4. .

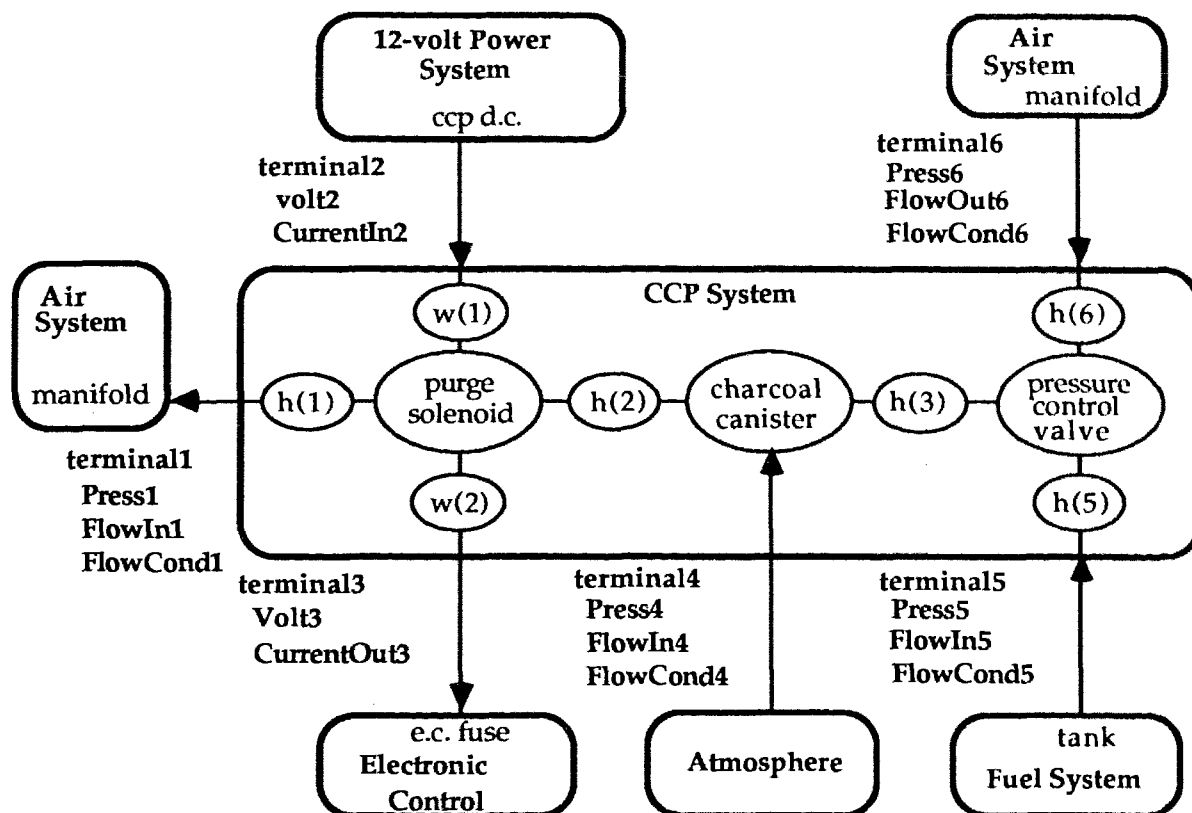


Figure 5-3
Charcoal Canister Purge System

The fuel system (Figure 5-4) distributes pressurized and filtered fuel from the fuel tank to the fuel injectors. The actual fuel system has six injectors, but for simplicity we have included only one. The electronic control system controls the flow of electricity to the injectors and the fuel pump. The fuel system shares interface variables with the electronic control system, engine mechanical system, CCP system, and air system. We did not model the engine mechanical system. Echidna code for the primitive sub-component models are found in appendices A.1. and A.2., and code for the fuel system is presented in appendices A.1. and A.4. .

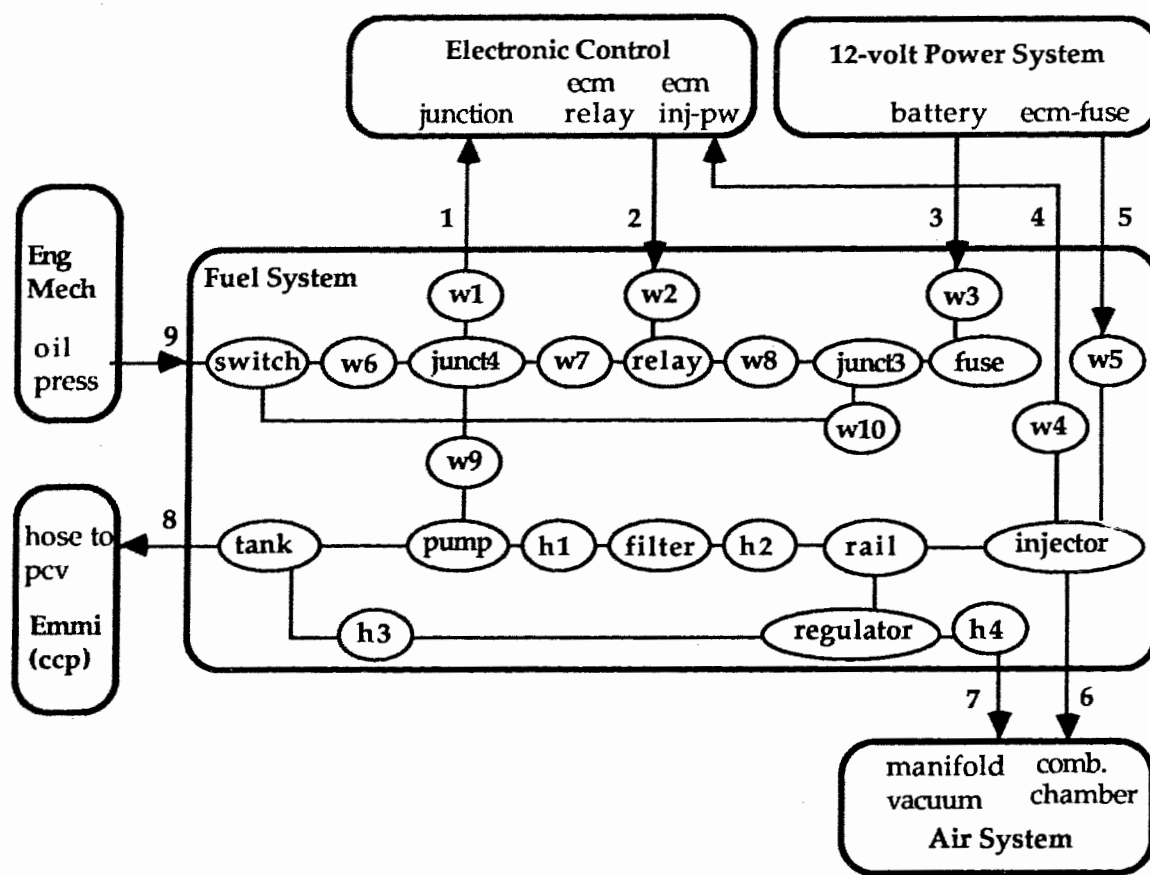


Figure 5-4
Fuel System

5.4. Simulated Diagnosis

We simulated a number of diagnostic sessions. The input files and output data for two of those simulations are shown in appendix B. We now give a general description of our diagnostic simulations followed by discussion of individual simulation results.

The first step in each simulation was loading the knowledge bases `component.kb`, `primitive.kb`, `compound.kb`, and `system.kb`. (for example see appendix B.1.1.). The artifact we are simulating is not always a complete vehicle. The first two simulations deal individually with the 12-volt power

system and the CCP system. The other systems contained in the loaded knowledge bases are not used in these simulations.

The second step of each simulation is to define, build, and name a model of the artifact, and to find candidates for that model. The input file calls the model the "top level object" because this is the object that we will communicate with through the Echidna interface. Since no measurement data has yet been given, no combination of component states leads to inconsistency. Since we do not wish to generate an exhaustive list of all possible state assignments, our diagnostic system will always look for healthy candidates first, and will always stop after finding a healthy candidate.

The third step of the simulated diagnostic session is to add measurement data (i.e., add evidence) to the diagnostic system, and to re-issue the command to find candidates. The system will re-generate the candidate found in the last step. If this candidate is consistent with the new data diagnostic reasoning will stop. If this candidate is inconsistent the system will attempt to find a healthy candidate that is consistent. If no healthy candidate can be found the system will exhaustively search for all single fault candidates, but without duplicating candidates that share the same culprit. Remember culprits are the components that are in a faulty state (i.e., Condition = bad).

Any additional steps are repeats of step three. Echidna does not forget the measurements from previous steps so the number of candidates (and culprits) will always decrease or stay the same.

We mentioned earlier that using the fail predicate to generate multiple solutions forced us to lose all solutions, including the last solution. The faulty solutions are printed to the screen, but then disappear. In this case, we must re-generate solutions to determine whether they are consistent with new evidence. After steps finding a healthy candidate it is possible to add new

evidence directly to the existing solution, but to be consistent we re-generated candidates after each measurement was added.

For each simulation we recorded elapsed time from the beginning of the session (i.e., before loading the knowledge bases) to the end of each step as described above. From this we calculated the incremental time for each step of the session. The incremental time is important because it is the time the technician will have to wait for a list of candidates. The time to load the knowledge bases is constant for all simulations, independent of the artifact we are diagnosing. Loading only those schemata needed would have been quicker in the cases where not all schemata were involved. Echidna offers an optimization option that compiles discrete constraints. We timed our simulated sessions with the optimization both on and off.

Table 5-1 shows the number of components in each of the systems we modelled. The power system, for instance, has one compound component (the ignition switch), eleven primitive components, and one system component (the power system itself).

	power1	ccp1	fuel1	engine1	vehicle1
no. of primitives	11	10	25	35	46
no. of compounds	1	0	0	0	1
no. of systems	1	1	1	3	5

Table 5-1
Number of Components

Our first simulated diagnostic session was for the 12-volt power system (see appendix B.1.1. for input data file, and B.1.2. for output file). The first line of entries in Table 5-2 shows the time taken to load the knowledge bases with the optimization on (43.5 seconds) and with the optimization off (4.8 seconds). As mentioned earlier knowledge bases were loaded for all our component models, so time was spent loading unnecessary schemata.

	no. of candidates	incremental time (seconds)		elapsed time (seconds)	
		optimized	non-optimized	optimized	non-optimized
load kb	-	43.5	4.8	43.5	4.8
no data	1 - healthy	2.5	1.9	46.0	6.7
measurements	1st 3 - faulty	3.9	6.3	49.9	13.0
	2nd 2 - faulty	1.5	2.9	51.4	15.9
	3rd 1 - faulty	2.7	1.8	54.1	17.7

Table 5-2

Power System Performance

The second line of entries in table 5-2 shows the incremental and elapsed times to define, name, and build the 12-volt power system and find a single healthy candidate. The incremental time was 2.5 seconds with the optimization on and 1.9 with the optimization off. Appendix B.1.2. shows the state and condition of all sub-components of the 12-volt power system. All are in state 0 and condition 0, which corresponds to a "good" state. Of course, no symptoms are present, because no measurements have been taken.

The third line of entries in table 5-2 represents the results of adding our first measurement (i.e., the signal was "off" and volt5 had a value of 6). This generated three faulty candidates in the times shown in incremental times of 3.9 and 6.3 seconds. The three culprits are the junction3 in state 1, the junction 3 in state 3, and the battery in state 1.

After the second measurement was added (i.e., the battery's voltage was 14) two candidates remained, the two involving the junction3. After the third measurement (junction3 volt2 = 14) only one candidate remained, junction 3 in state 3.

Our second simulated diagnostic session was for the Charcoal canister purge system (see appendix B.2.1. for input data file, and B.2.2. for output file). The first line of entries in table 5-3 shows the times it took to load the knowledge bases with the optimization both on and off, which, as expected are identical to the times in Table 5-1 above.

	no. of candidates	incremental time (seconds)		elapsed time (seconds)		
		optimized	non-optimized	optimized	non-optimized	
load kb	-	43.5	4.8	43.5	4.8	
no data	1 - healthy	1.1	1.1	44.6	5.9	
measurements	1st	1 - healthy	0.4	0.6	45.0	6.5
	2nd	6 - faulty	8.4	9.4	53.4	15.9
	3rd	6 - faulty	8.9	9.2	62.3	25.1
	4th	2 - faulty	11.9	12.7	74.2	37.8

Table 5-3
CCP System Performance

The second line of entries table 5-3 shows the incremental and elapsed time to define, name, and build the CCP system and find a single healthy candidate. The incremental time was 1.1 seconds with the optimization on and off. This step was slightly quicker than for the power system because the power system has a compound component. The CCP and power systems have similar numbers of primitive components (11 versus 10). Appendix B.2.2. shows the state and condition of all sub-components of the CCP system. All are in state 0 and condition 0, which corresponds to a "good" state. Of course, no symptoms are present.

The third line of entries in table 5-3 represents the results of adding our first measurement (volt2 had a value of 14 and volt3 had a value of 0). This data was not inconsistent with the healthy candidate generated in the last step, so reasoning stopped after this candidate was re-generated.

The fourth line of entries in table 5-3 represents the results of adding our second measurement (press1 = -1 and flowOut1 = 2). This data resulted in six faulty candidates.

The fifth line of entries in table 5-3 represents the results of adding our third measurement (press6 = -1). This data did not reduce the number of candidates, so was a poor choice of measurement. Ideally the diagnostic system would advise which measurements will best discriminate between competing candidates.

The sixth line of entries in table 5-3 represents the results of adding our fourth measurement (press5 = 2 and flowIn5 = 2). This data eliminated all but two faulty candidates, one implicating the hose1 and the other implicating the solenoid. Another measurement would be needed to identify which of these is the actual candidate.

The final two simulations were for the engine (which was composed of the fuel system and the CCP system) and for the vehicle (which was composed of the engine and the 12-volt power system). Unfortunately, we did not have time to generate test data that would allow us to incrementally eliminate faulty candidates. We were able to build models of these systems and find a healthy candidate, the incremental and elapsed times for which are shown in Table 5-4 and Table 5-5.

	no. of candidates	incremental time (seconds)		elapsed time (seconds)	
		optimized	non-optimized	optimized	non-optimized
load kb	-	43.5	4.8	43.5	4.8
build	1 - healthy	5.4	6.4	48.9	11.2

Table 5-4
Engine Performance

	no. of candidates	incremental time (seconds)		elapsed time (seconds)	
		optimized	non-optimized	optimized	non-optimized
load kb	-	43.5	4.8	43.5	4.8
build	1 - healthy	8.5	11.1	52.0	15.9

Table 5-5
Vehicle Performance

5.4.1. Analysis of Performance

The optimization technique is designed to speed up reasoning using discrete constraints. The penalty for this speed increase is slower loading of knowledge

bases. In our case, especially in the first two simulations, we compiled many more constraints than were actually used. While the diagnostic steps were between 0 to 50% faster for the optimized version, the loading stage was approximately ten times slower. In the first two simulations the complete non-optimized diagnostic simulation took less time than the optimized knowledge base loading. Our vehicle model had a total of 52 components. If a complete vehicle model had, say, 1000 components, and a linear relationship between loading and the number of components, then the loading time would be approximately 10 minutes for the optimized version and approximately 2 minutes for the non-optimized version. The possibility of dynamically loading knowledge bases (i.e., automatically loading knowledge bases as they are needed) may need to be explored.

Step two for our vehicle model took 8.5 seconds for the optimized run and 11.1 for the non-optimized run. This step always finds one healthy candidate. Our vehicle model represents a small fraction of a complete vehicle. Even a linear time increase with the number of components would result in minutes of waiting time for step two.

The first two simulations took even longer on the steps following step two. Part of the problem was that we were regenerating candidates at each step rather than continuing reasoning from the last step. Ideally Echidna would maintain a set of candidates between steps and, as new measurements are obtained, continue reasoning on each candidate in parallel until only one remains.

5.5. Suggested Implementation Improvements

To encourage modularity and make the knowledge base easier to maintain, extend, and debug, global constraints should be explicitly represented as such, either in a separate global schema, or as global methods outside of any declared

schema. The single fault rule is a global constraint, but is hidden within the inheritance hierarchy. For example, switching to (or adding) a double fault rule would be much easier if global constraints were explicit.

The assignment of condition to compound components could be automatically accomplished through a method in the compound schema rather than coded in method `mode/0` in each compound component. Based on the sub-component list a domain for the condition could be dynamically created. For example, the "Condition" might be stored in a vector with the first element having the value good or bad, and the second element indicating which, if any, components are bad. This could be expanded to multiple fault candidates by allowing the second element to be a list of indeterminate length.

To simplify coding, the domains of interface variables were declared in advance, and were fixed for different units (e.g., all voltages were declared to have the domain of -20 to +20 volts). Realistically, different physical components represented by the same behavioural model would have different domains for their interface variables.

We have separate models for junctions of three and four electrical connections. Junctions of any number of conductors can be described by two laws, Kirchoff's current law and Kirchoff's voltage law. We should have a general junction model with a variable number of conductors.

Likewise, our model of a rail in a fuel injection system is more generally a model of a junction of three fluid conduits. We may be better off with a model called "fluidJunction" which has a variable number of branches (each with a variable size). We would create a rail with tree branches of sizes 1.5 1.0 and 0.5 by declaring "rail isa junction, rail:define(3,(1.5 1 .5))".

Both of the last two improvements hint at a stronger one. We could model general physical component types such as valves (switches, solenoids,

regulators), junctions, reservoirs (fuel tanks, charcoal canisters), sources, conduits (wires, hoses). For example, a simple switch could be modelled as an instance of an electrical valve with a binary (on/off) control setting.

6. Conclusions

The application of model-based diagnosis to automotive engines, and mechanical systems in general, has not been widely reported. A number of authors have developed prototypes of rule-based automobile diagnosis expert systems.

Engine modelling is presently pursued for two different purposes - engine design and engine control. Such models are not developed with expert systems implementation in mind. Diagnosis is concerned with only those variables which can be changed after the vehicle is in operation (i.e., calibration, adjustments, replacement of components, etc.), while engine design models are concerned with quantitative variables that can be changed before engine components are made (e.g., valve seat angles, displacement, number of cylinders, etc.). Many engine design models are computationally expensive because they calculate precise quantitative answers from complicated mathematical equations. Control models were traditionally developed solely by matching outputs with their causal inputs, although physical models are now deriving behaviour from underlying physical principles. Still, all control models still suffer a significant reliance on engine testing for estimation of empirical parameters.

The most popular domain for the study of model-based diagnosis is electronic circuits because this domain is blessed with a close relationship between structure and function. Knowing a circuit's structure (i.e., the arrangement of resistors, capacitors, transistors, etc.) a reasonably accurate model of behaviour can be developed relatively easily. The ease of modelling electronic circuit behaviour allows researchers to concentrate on diagnostic theory rather than complexities in modelling.

A good example of a model-based diagnosis is the General Diagnostic Engine (GDE) [deKleer 87], which can incrementally diagnose multiple faults, is domain independent, and is able to propose measurements to localize faults. GDE incorporated an assumption-based multiple context reason maintenance system, Bayesian probability methods to rank diagnoses, and decision theory to estimate the next best measurement. GDE was superseded by SHERLOCK [deKleer 89] which allowed more modelling flexibility, reduced physically impossible diagnoses, and provided heuristics to minimize combinatorial explosion.

A different approach to multiple fault model-based diagnosis was provided by the Bayesian Belief Network [Pearl 88]. Here, the model is a causal network of components that send messages to their neighbours in the form of probability distributions. New evidence results in two passes of messages through the network, after which all components will be in their most likely state.

Our qualitative modelling strategy builds on that of SHERLOCK. We add specialization and composition hierarchies, and different types of component models. The specialization hierarchy allows components of the model to inherit behaviour from more general components. The two basic categories of component models are primitive component models and composite component models.

Primitive component models are at the lowest level of abstraction. They represent fallible physical components that are replaced as single units. These models have interface variables they share with the component models they are connected to, and a finite and exhaustive set of behavioural modes which relate those variables. Some of the modes represent healthy operation, and some represent faulty operation.

Composite models are divided into compound and system models. System models represent a group of physical sub-components that work together to

perform a task (e.g. the fuel system). The behaviour of a system component model is not described explicitly, rather it is derived from that of its sub-components. If any of the sub-components have failed then the system has also failed.

Compound component models are similar to system models because they have a number of sub-components, but differ in that they represent a single physical component. Complicated components such as ignition switches are more easily represented as a combination of simpler switches and junctions. An ignition switch could be modelled as a primitive component, but defining all the behavioural modes would be tedious. Compound components are replaced as a single unit.

A composition hierarchy gives an intuitive organization to the model. In this tree structure the nodes represent components of the model, and the arcs represent part-of links. The root node is the complete artifact (e.g., engine or vehicle) and the leaf nodes represent physical fallible components. All nodes other than the leaf nodes are groups of components that cooperate to perform a task.

At this point our model is not able to reason with time. The behaviour of some components varies with time, i.e., outputs at a later time depend on the inputs at an earlier time. This time delay problem can be neglected in diagnosing combinatorial digital circuits, hence the interest in them as trial domains for general diagnostic systems. Modelling mechanical systems with time delays is more complicated.

Our model does not store information necessary to generate candidates in a ranked order (i.e., the most likely first). Presently candidates are found in an order depending on the syntactic ordering of the knowledge base. Because of this we limited our model to single fault candidates, which vastly reduces the number of possible candidates, and forces all candidates to be reasonably likely.

Adding an unknown mode to each primitive component would make our model competent for single fault diagnoses. However, we chose to forgo competence because we assume the unknown mode is much less likely than those we explicitly represented, and would result in less likely candidates.

Our prototype model was successfully implemented in the Echidna constraint reasoning system running on a Sun SPARCstation 1, and had schemata representing portions of a vehicle, engine, fuel system, charcoal canister purge system, and 12-volt power system. Echidna was being developed at the same time as our model, and many advanced features that could improve knowledge representation and reasoning efficiency have since become available. Performance was adequate for our prototype model, taking an average of approximately 5 seconds after new measurement data was received to generate a new list of candidates. However, a model of a full engine or vehicle would be much larger, and response times would be on the order of minutes. Part of the reason for the slow speed was the inability of the early version of Echidna to generate and retain multiple solutions to a single diagnostic query.

Appendix A - Knowledge Bases

A.1. Component Knowledge Base

```
%-----  
%  
% Automotive Diagnosis  
% charlie/diagnosis/new/component.kb  
%  
% This file has schemas for the top levels of the inheritance  
% hierarchy - component, primitive, composite, compound, and  
% system. Primitive and composite inherit from component, and  
% compound and system inherit from composite.  
%  
%-----  
%  
% user defined Types  
%  
%-----  
  
signalType = {off,on}.  
conditionType = {good,bad}.  
  
%  
% All voltages will have integer (discrete) values between -20 and +20  
%  
  
voltageRange = {-20..20}.  
  
%  
% All currents will have integer (discrete) values between -10 and +10. This  
% is a simplifying and time saving assumption. In reality some automotive  
% components carry much more than 10 amperes  
%  
  
currentRange = {-10..10}.  
  
%  
% All pressures will have integer (discrete) values between -5 and +5  
%  
  
pressRange = {-5..5}.
```

```

%
% All flow rates will have integer (discrete) values between -5 and +5
%

flowRateRange = {-5..5}.

%-----
%
% Schema Definitions
%
%-----

%-----
%
% Schema component describes a general component. Other more specific
% components will be derived from component
%
%-----

schema component
{
    %
    % Schema instance variables
    %

    Name.
    integer State.
    integer Condition.
    component Clist.
    PrintList.

    %
    % accessors for schema instance variables
    %

    condition(Condition).
    name(Name).
    state(State).
    printList(PrintList).

    %
    % component methods
    %

    %
    % mode/0 will be redefined in more specific components
    %

    mode.

```

```

%
% build/0 unifies Printlist with the Name State and Condition
% of the component
%

build:- PrintList = [Name,State,Condition].

%
% healthyCandidate(SubComponentList) unifies Condition with 0,
% and issues the goal mode
%

healthyCandidate(SubComponentList):-
    condition(0),
    mode.

%
% singleFaultCandidate(SubComponentList,Flag) issues the goal
% mode
%

singleFaultCandidate(SubComponentList,Flag):- mode.

%
% findCulprit(_Culprit) unifies Culprit with State if
% Condition = 1
%

order findCulprit.

findCulprit(C1,State):- Condition == 1.

findCulprit(C1,Culprit):- Condition == 0.
)

%-----
%
% Schema primitive is derived from component. Primitive components
% correspond to actual physical components of the engine.
%
%-----

schema primitive:component
{
    %
    % primitive methods
    %

    %
    % initialization - primitive components have Condition 0 (good)
    % or 1 (bad).

```

```

%
%
% :- Condition = {0,1} _ .
}

%-----
%
% Schema composite is derived from component. Composite components derive
% their behaviour from their sub-components.
%
%-----

schema composite:component
{
%
% composite methods
%

%
% buildSubComps(SubComponentList) recursively builds the
% sub-components in the list.
%

buildSubComps([]).

buildSubComps([component Hcomp | Tcomps]):-
    Hcomp:build,
    buildSubComps(Tcomps).

%
% compositeState(SubComponentList,OList,State) recursively
% assembles the State of a composite component from the States
% of the sub-components in SubComponentList.
%

compositeState([],State,State).

compositeState([component Hcomp | Tcomps],OldList,State):-
    Hcomp:state(Cstate),
    compositeState(Tcomps,[Cstate | OldList],State).

}

%-----
%
% Schema compound is derived from composite. Compound components are
% modelled as a group of components, but correspond to a single physical
% component of an engine.
%
%-----

```

```

schema compound:composite
{
    %
    % compound methods
    %

    :- Condition = {0,1} _ .

    %
    % build/0 issues goals buildSubComps/1, compositeState/3,
    % nameAndStateList/3, and unifies PrintList with Name, NSList
    % from nameAndStateList/3, and Condition.
    %

    build:- buildSubComps(Clist),
            compositeState(Clist,[],State),
            nameAndStateList(Clist,[],NSList),
            PrintList = [Name,NSList,Condition].

    %
    % nameAndStateList(SubComponentList, OldList, NandSList)
    % assembles in NandSList a list of the Name and State of each
    % of the components in SubComponentList.
    %

    nameAndStateList([],NSList,NSList).

    nameAndStateList([component Hcomp | Tcomps],OldList,NSList):-
        Hcomp:name(Cname),
        Hcomp:state(Cstate),
        nameAndStateList(Tcomps,[[Cname,Cstate] | OldList],NSList).

}

%-----
%
% Schema system is derived from composite. System components are groups
% of connected components.
%-----

schema system:composite
{
    %
    % system methods
    %

    %
    % build/0 issues goals buildSubComps/1, compositeState/3,
    % systemCondition/3, and partOfPrintList/3 and unifies PrintList
    % with Name and PartOfList from partOfPrintList/3.

```

```

%

build:- buildSubComps(Clist),
        compositeState(Clist,[],State),
        systemCondition(Clist,[],Condition),
        partOfPrintList(Clist,[],PartOfList),
        PrintList = [Name,PartOfList].

%
% partOfPrintList(SubComponentList,OldList,PartOfList)
% assembles in PartOfList a list of the Name, State, and
% Condition of each of the components in SubComponentList.
%

partOfPrintList([],PartOfList,PartOfList).

partOfPrintList([component Hcomp | Tcomps],OldList,PartOfList):-
    Hcomp:printList(PList),
    partOfPrintList(Tcomps,[PList | OldList],PartOfList).

%
% systemCondition(SubComponentList,OldList,Condition) recursively
% assembles the Condition of a composite component from the
% Conditions of the sub-components in SubComponentList.
%

systemCondition([],Condition,Condition).

systemCondition([component Hcomp | Tcomps],OldList,Condition):-
    Hcomp:condition(Ccondition),
    systemCondition(Tcomps,[Ccondition | OldList],Condition).

%-----

%
% findCandidates/0 first tries findHealthyCandidate, then tries
% findSingleFaultCandidate. A candidate is an assignment of modes
% to all primitive components
%

order findCandidates.

findCandidates:- findHealthyCandidate.

findCandidates:- findSingleFaultCandidates.

%
% findHealthyCandidate/0 issues goal healthyCandidate/1, and then
% prints PrintList on screen.
%
```



```

findSingleFaultCandidates:-
    singleFaultCandidate(Clist,0)^
    print("\n$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n\n"),
    printRoutine(PrintList)^
    print("\nlooking for more single fault candidates\n"),
    findCulprit(Clist,Culprit),
    fail(Culprit).

findSingleFaultCandidates:-
    print("\nno more single fault candidates\n\n"),
    print("$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n\n").

%
% singleFaultCandidate(SubComponentList,Flag) recursively sends
% the message singleFaultCandidate/2 to each component in
% SubComponentList, issues goals conditionSummary/2 and
% singleFault/3.
%
singleFaultCandidate([],1).

singleFaultCandidate([component Hcomp | Tcomps],Inflag):-
    Hcomp:singleFaultCandidate(Clist,0),
    Hcomp:condition(Ccond),
    conditionSummary(Ccond,CondSummary),
    singleFault(Inflag,Outflag,CondSummary),
    singleFaultCandidate(Tcomps,Outflag).

%
% findCulprit(SubComponentList,Culprit) recursively searches
% SubComponentList to find the component which is in a faulty
% mode and returns that components State in Culprit
%
findCulprit([],Culprit).

findCulprit([component Hcomp | Tcomps],Culprit):-
    Hcomp:findCulprit(Clist,Culprit),
    findCulprit(Tcomps,Culprit).

%
% conditionSummary(ComponentCondition,Summary) makes
% Summary
% and ComponentCondition equal for primitive components, and
% issues goal member/2 if ComponentCondition is a List
%
order conditionSummary.

conditionSummary(0,0).

```

```
conditionSummary(1,1).
conditionSummary(Ccond,1):- member(1,[Ccond]).

%
% member(a,List) succeeds if 1 is a member of List. List can
% have an arbitrary structure (i.e., lists within lists).
%

order member.

member(X,[X|_]).
member(X,[_|_]):- member(X,_).
member(X,[_|_]):- member(X,_).

%
% singleFault(InFlag,OutFlag,Condition) fails if both OutFlag
% and Condition are 1.
%

singleFault(0,0,0).
singleFault(0,1,1).
singleFault(1,1,0).
```

}

A.2. Primitive Knowledge Base

```

%-----
%
% Automobile Diagnosis
% charlie/diagnosis/new/primitive.kb
%
% This file contains schemas for primitive component models. All inherit
% methods from schema primitive in file component.kb
%
%-----

%-----
%
% Schema fuse is derived from primitive. This is a model of a simple
% electrical fuse
%
%-----

schema fuse:primitive
{
    %
    % Schema instance variables
    %

    voltageRange Volt1. currentRange CurrentIn1.
    voltageRange Volt2. currentRange CurrentOut2.

    %
    % accessors for schema instance variables
    %

    terminal1(Volt1,CurrentIn1).
    terminal2(Volt2,CurrentOut2).
    volt1(Volt1). currentIn1(CurrentIn1).
    volt2(Volt2). currentOut2(CurrentOut2).

    %
    % fuse methods
    %

    %
    % initialization -fuse has state 0 or 1
    %

    :- State = {0,1} _.
```

```

%
% mode/0 defines behavioral modes of fuse
%

order mode.

mode:- % good state
    CurrentOut2 ::= CurrentIn1,
    Volt2 ::= Volt1,
    State ::= 0,
    Condition ::= 0.

mode:- % bad state - fuse blown
    CurrentOut2 ::= 0,
    CurrentIn1 ::= 0,
    State ::= 1,
    Condition ::= 1.
}

%-----
%
% Schema junction3 is derived from primitive. This is a model of a
% junction of three electrical conductors.
%
%-----

schema junction3:primitive
{
    %
    % Schema instance variables
    %

    voltageRange Volt1. currentRange CurrentIn1.
    voltageRange Volt2. currentRange CurrentIn2.
    voltageRange Volt3. currentRange CurrentIn3.

    %
    % accessors for schema instance variables
    %

    terminal1(Volt1,CurrentIn1).
    terminal2(Volt2,CurrentIn2).
    terminal3(Volt3,CurrentIn3).
    volt1(Volt1). currentIn1(CurrentIn1).
    volt2(Volt2). currentIn2(CurrentIn2).
    volt3(Volt3). currentIn3(CurrentIn3).

    %
    % junction3 methods
    %

```

```

%
% initialization - junction3 has state 0 to 4
%

:- State = {0..4} _;

%
% mode/0 defines behavioral modes of junction3
%

order mode.

mode:- % good condition
    Volt1 == Volt2,
    Volt2 == Volt3,
    CurrentIn1 + CurrentIn2 + CurrentIn3 == 0,
    State == 0,
    Condition == 0.

mode:- % open circuit on 1
    CurrentIn2 + CurrentIn3 == 0,
    CurrentIn1 == 0,
    Volt2 == Volt3,
    State == 1,
    Condition == 1.

mode:- % open circuit on 2
    CurrentIn1 + CurrentIn3 == 0,
    CurrentIn2 == 0,
    Volt1 == Volt3,
    State == 2,
    Condition == 1.

mode:- % open circuit on 3
    CurrentIn1 + CurrentIn2 == 0,
    CurrentIn3 == 0,
    Volt1 == Volt2,
    State == 3,
    Condition == 1.

mode:- % shorted circuit
    Volt1 == Volt2,
    Volt2 == Volt3,
    CurrentIn1 + CurrentIn2 + CurrentIn3 == 0,
    State == 4,
    Condition == 1.

```

```

}

```

```

%-----
%

```

```

% Schema switch is derived from primitive. This is a model of a simple
% single post single throw electrical switch
%
%-----

```

```

schema switch:primitive
{

```

```

    %
    % Schema instance variables
    %
    voltageRange Volt1. currentRange CurrentIn1.
    voltageRange Volt2. currentRange CurrentOut2.
    signalType Signal.

```

```

    %
    % accessors for schema instance variables
    %

```

```

    terminal1(Volt1,CurrentIn1).
    terminal2(Volt2,CurrentOut2).
    volt1(Volt1). currentIn1(CurrentIn1).
    volt2(Volt2). currentOut2(CurrentOut2).
    signal(Signal).

```

```

    %
    % switch methods
    %

```

```

    %
    % initialization - switch has state 0 to 3
    %

```

```

:- State = {0..3} _ .

```

```

    %
    % mode/0 defines behavioral modes of switch
    %

```

```

order mode.

```

```

mode:- % Off condition
    Signal == off,
    CurrentIn1 == 0,
    CurrentOut2 == 0,
    State == 0,
    Condition == 0.

```

```

mode:- % On condition
    Signal == on,
    CurrentIn1 == CurrentOut2,

```

```

    Volt1 ::= Volt2,
    State ::= 1,
    Condition ::= 0.

mode:- % short circuit across throw
    CurrentIn1 ::= CurrentOut2,
    Volt1 ::= Volt2,
    State ::= 2,
    Condition ::= 1.

mode:- % open circuit across throw
    CurrentIn1 ::= 0,
    CurrentOut2 ::= 0,
    State ::= 3,
    Condition ::= 1.
}

%-----
%
% Schema wire is derived from primitive. This is a model of a simple
% electrical conductor
%
%-----

schema wire:primitive
{
    %
    % Schema instance variables
    %

    voltageRange Volt1. currentRange CurrentIn1.
    voltageRange Volt2. currentRange CurrentOut2.

    %
    % accessors for schema instance variables
    %

    terminal1(Volt1,CurrentIn1).
    terminal2(Volt2,CurrentOut2).
    volt1(Volt1). currentIn1(CurrentIn1).
    volt2(Volt2). currentOut2(CurrentOut2).

    %
    % wire methods
    %

    %
    % initialization - wire has state 0 to 3
    %

    :- State = {0..3} _.
```

```

%
% mode/0 defines behavioral modes of wire
%

```

```

order mode.

```

```

mode: % good state
      Volt1 == Volt2,
      CurrentIn1 == CurrentOut2,
      State == 0,
      Condition == 0.

```

```

mode:- % shorted state
      Volt1 == Volt2,
      State == 1,
      Condition == 1.

```

```

mode:- % open state
      CurrentIn1 == 0,
      CurrentOut2 == 0,
      State == 2,
      Condition == 1.

```

```

}

```

```

%-----
%
% Schema battery is derived from primitive. This is an extremely simple
% and unrealistic model of a 12 volt automotive battery. This model
% assumes the battery is a source, i.e., it has only one terminal at
% which a voltage and current can be measured. A more realistic model
% would be required if the battery was to be connected to a model of
% the ground circuits.
%
%-----

```

```

schema battery:primitive

```

```

{

```

```

  %
  % Schema instance variables
  %

```

```

  voltageRange Volt. currentRange CurrentOut.

```

```

  %
  % accessors for schema instance variables
  %

```

```

  terminal(Volt,CurrentOut).
  volt(Volt). currentOut(CurrentOut).

```



```

%
% battery methods
%

%
% initialization - battery has state 0 to 2
%

:- State = {0..2} _ .

%
% mode/0 defines behavioral modes of battery
%

order mode.

mode:- % good state, voltage between 9 and 16 volts
      Volt >= 9,
      Volt <= 16,
      State == 0,
      Condition == 0.

mode:- % bad state, undercharged
      Volt < 9,
      State == 1,
      Condition == 1.

mode:- % bad state, overcharged
      Volt > 16,
      State == 2,
      Condition == 1.
}

%-----
%
% Schema hose is derived from primitive. This is a model of a simple
% hose
%
%-----

schema hose:primitive
{
  %
  % Schema instance variables
  %

  pressRange Press1. flowRateRange FlowIn1. conditionType FlowCond1.
  pressRange Press2. flowRateRange FlowOut2. conditionType FlowCond2.

  %
  % accessors for schema instance variables

```

```

%

terminal1(Press1,FlowIn1,FlowCond1).
terminal2(Press2,FlowOut2,FlowCond2).

press1(Press1). flowIn1(FlowIn1). flowCond1(FlowCond1).
press2(Press2). flowOut2(FlowOut2). flowCond2(FlowCond2).

%
% hose methods
%

%
% initialization - hose has state 0 to 2
%

:- State = {0..2} _ .

%
% mode/0 defines behavioral modes of hose
%

order mode.

mode:- % good state
      FlowIn1 == FlowOut2,
      Press1 == Press2,
      FlowCond1 == FlowCond2,
      State == 0,
      Condition == 0.

mode:- % leaking state
      FlowOut2 =\= FlowIn1,
      Press1 == Press2,
      State == 1,
      Condition == 1.

mode:- % blocked state
      FlowOut2 == 0,
      FlowIn1 == 0,
      State == 2,
      Condition == 1.
}

%-----
%
% Schema canister is derived from primitive. This is a model of a simple
% charcoal canister for a charcoal canister purge system
%
%-----

```

```

schema canister:primitive
{
    %
    % Schema instance variables
    %

    pressRange Press1. flowRateRange FlowOut1. conditionType FlowCond1.
    pressRange Press2. flowRateRange FlowIn2. conditionType FlowCond2.
    pressRange Press3. flowRateRange FlowIn3. conditionType FlowCond3.

    %
    % accessors for schema instance variables
    %

    terminal1(Press1,FlowOut1,FlowCond1).
    terminal2(Press2,FlowIn2,FlowCond2).
    terminal3(Press3,FlowIn3,FlowCond3).

    press1(Press1). flowOut1(FlowOut1). flowCond1(FlowCond1).
    press2(Press2). flowOut2(FlowIn2). flowCond2(FlowCond2).
    press3(Press3). flowOut3(FlowIn3). flowCond3(FlowCond3).

    %
    % canister methods
    %

    %
    % initialization - canister has state 0 to 5
    %

    :- State = {0..5} _ .

    %
    % mode/0 defines behavioral modes of canister
    %

    order mode.

    mode:- % good state, not storing nor purging
        FlowOut1 == 0,
        FlowIn3 == 0,
        FlowIn2 == 0,
        Press1 == Press3,
        Press1 == Press2,
        State == 0,
        Condition == 0.

    mode:- % good state, storing but not purging
        FlowOut1 == 0,
        FlowIn2 == 0,
        FlowIn3 > 0,

```

```

    Press3 ::= Press1,
    State ::= 1,
    Condition ::= 0.

mode:- % good state, purging but not storing
    FlowIn3 ::= 0,
    FlowOut1 ::= FlowIn2,
    Press1 < Press2,
    Press1 ::= Press3,
    State ::= 2,
    Condition ::= 0.

mode:- % bad state, blocked inlet no. 2
    FlowOut1 ::= 0,
    FlowIn2 ::= 0,
    State ::= 3,
    Condition ::= 1.

mode:- % bad state, blocked inlet no. 3
    FlowIn3 ::= 0,
    State ::= 4,
    Condition ::= 1.

mode:- % bad state, leaking
    FlowOut1 =\= FlowIn2,
    State ::= 5,
    Condition ::= 1.
}

%-----
%
% Schema pcv is derived from primitive. This is a model of a simple
% pressure control valve as used in the charcoal canister purge system
%
%-----

schema pcv:primitive
{
    %
    % Schema instance variables
    %

    pressRange Press1. flowRateRange FlowOut1. conditionType FlowCond1.
    pressRange Press2. flowRateRange FlowIn2. conditionType FlowCond2.
    pressRange Press3. flowRateRange FlowIn3. conditionType FlowCond3.

    %
    % accessors for schema instance variables
    %

    terminal1(Press1,FlowOut1,FlowCond1).

```

```

terminal2(Press2,FlowIn2,FlowCond2).
terminal3(Press3,FlowIn3,FlowCond3).

press1(Press1). flowOut1(FlowOut1). flowCond1(FlowCond1).
press2(Press2). flowIn2(FlowIn2). flowCond2(FlowCond2).
press3(Press3). flowIn3(FlowIn3). flowCond3(FlowCond3).

%
% pcv methods
%

%
% initialization - pcv has state 0 to 6
%

:- State = {0..6} _ .

%
% mode/0 defines behavioral modes of pcv
%

order mode.

mode:- % Valve is closed by vacuum on terminal 1
    Press1 < 0,
    FlowOut1 == 0,
    FlowIn2 == 0,
    State == 0,
    Condition == 0.

mode:- % Valve is opened by pressure on terminal 2
    Press1 >= 0,
    Press2 > 4,
    FlowOut1 == FlowIn2,
    State == 1,
    Condition == 0.

mode:- % Valve is opened by vacuum on terminal 3
    Press1 >= 0,
    Press3 < 0,
    Press1 == Press2,
    FlowOut1 == FlowIn2,
    State == 2,
    Condition == 0.

mode:- % Valve blocked
    FlowIn2 == 0,
    FlowOut1 == 0,
    State == 3,
    Condition == 1.

```

```

mode:- % Valve stuck open
    Press1 == Press2,
    FlowOut1 == FlowIn2,
    State == 4,
    Condition == 1.

mode:- % Valve restricted
    Press1 == Press2,
    FlowOut1 == FlowIn2,
    State == 5,
    Condition == 1.

mode:- % Valve leaking
    FlowOut1 == FlowIn2,
    State == 6,
    Condition == 1.
}

%-----
%
% Schema solenoid is derived from primitive. This is a model of a simple
% electrical solenoid valve used to control fluid flow
%
%-----

schema solenoid:primitive
(
    %
    % Schema instance variables
    %

    pressRange Press1. flowRateRange FlowOut1. conditionType FlowCond1.
    pressRange Press2. flowRateRange FlowIn2. conditionType FlowCond2.
    voltageRange Volt3. currentRange CurrentIn3.
    voltageRange Volt4. currentRange CurrentOut4.

    %
    % accessors for schema instance variables
    %

    terminal1(Press1,FlowOut1,FlowCond1).
    terminal2(Press2,FlowIn2,FlowCond2).
    terminal3(Volt3,CurrentIn3).
    terminal4(Volt4,CurrentOut4).

    press1(Press1). flowOut1(FlowOut1). flowCond1(FlowCond1).
    press2(Press2). flowIn2(FlowIn2). flowCond2(FlowCond2).
    volt3(Volt3). currentIn3(CurrentIn3).
    volt4(Volt4). currentOut4(CurrentOut4).

    %

```

```

% solenoid methods
%

%
% initialization - solenoid has state 0 to 5
%

:- State = {0..5} _ .

%
% mode/0 defines behavioral modes of solenoid
%

order mode.

mode:- % good state (closed solenoid)
    FlowOut1 == 0,
    FlowIn2 == 0,
    Volt3 - Volt4 >= 9,
    Volt3 - Volt4 <= 16,
    CurrentIn3 == CurrentOut4,
    State == 0,
    Condition == 0.

mode:- % good state (open solenoid)
    FlowOut1 == FlowIn2,
    Press1 == Press2,
    CurrentIn3 == 0,
    CurrentOut4 == 0,
    Volt3 == Volt4,
    State == 1,
    Condition == 0.

mode:- % blocked state
    FlowOut1 == 0,
    FlowIn2 == 0,
    State == 2,
    Condition == 1.

mode:- % stuck open state
    FlowOut1 == FlowIn2,
    Press1 == Press2,
    State == 3,
    Condition == 1.

mode:- % restricted state
    FlowOut1 == FlowIn2,
    Press1 \= Press2,
    State == 4,
    Condition == 1.

```

```

mode:- % leaking state
      FlowIn2 =\= FlowOut1,
      State ::= 5,
      Condition ::= 1.
}

%-----
%
% Schema junction4 is derived from primitive. This is a model of a simple
% electrical junction with four branches
%
%-----

schema junction4:primitive
{
  %
  % Schema instance variables
  %

  voltageRange Volt1. currentRange CurrentIn1.
  voltageRange Volt2. currentRange CurrentIn2.
  voltageRange Volt3. currentRange CurrentIn3.
  voltageRange Volt4. currentRange CurrentIn4.

  %
  % accessors for schema instance variables
  %

  terminal1(Volt1,CurrentIn1).
  terminal2(Volt2,CurrentIn2).
  terminal3(Volt3,CurrentIn3).
  terminal4(Volt4,CurrentIn4).

  volt1(Volt1).currentIn1(CurrentIn1).
  volt2(Volt2).currentIn2(CurrentIn2).
  volt3(Volt3).currentIn3(CurrentIn3).
  volt4(Volt4).currentIn4(CurrentIn4).

  %
  % junction4 methods
  %

  %
  % initialization - junction4 has state 0 to 5
  %

  :- State = {0..5} _ .

  %
  % mode/0 defines behavioral modes of solenoid
  %

```


order mode.

mode:- % good condition

Volt1 ::= Volt2,
 Volt2 ::= Volt3,
 Volt3 ::= Volt4,
 CurrentIn1 + CurrentIn2 + CurrentIn3 + CurrentIn4 ::= 0,
 State ::= 0,
 Condition ::= 0.

mode:- % open circuit on 1

CurrentIn2 + CurrentIn3 + CurrentIn4 ::= 0,
 CurrentIn1 ::= 0,
 Volt2 ::= Volt3,
 Volt3 ::= Volt4,
 State ::= 1,
 Condition ::= 1.

mode:- % open circuit on 2

CurrentIn1 + CurrentIn3 + CurrentIn4 ::= 0,
 CurrentIn2 ::= 0,
 Volt1 ::= Volt3,
 Volt3 ::= Volt4,
 State ::= 2,
 Condition ::= 1.

mode:- % open circuit on 3

CurrentIn1 + CurrentIn2 + CurrentIn4 ::= 0,
 CurrentIn3 ::= 0,
 Volt1 ::= Volt2,
 Volt2 ::= Volt4,
 State ::= 3,
 Condition ::= 1.

mode:- % open circuit on 4

CurrentIn1 + CurrentIn2 + CurrentIn3 ::= 0,
 CurrentIn4 ::= 0,
 Volt1 ::= Volt2,
 Volt2 ::= Volt3,
 State ::= 4,
 Condition ::= 1.

mode:- % shorted circuit

Volt1 ::= Volt2,
 Volt2 ::= Volt3,
 Volt3 ::= Volt4,
 CurrentIn1 + CurrentIn2 + CurrentIn3 + CurrentIn4 = \= 0,
 State ::= 5,
 Condition ::= 1.

```

%-----
%
% Schema relay is derived from primitive. This is a model of a simple
% electrical relay
%
%-----

```

```

schema relay:primitive
{

```

```

    %
    % Schema instance variables
    %

    voltageRange Volt1. currentRange CurrentIn1.
    voltageRange Volt2. currentRange CurrentOut2.
    voltageRange Volt3. currentRange CurrentIn3.
    voltageRange Volt4. currentRange CurrentOut4.

```

```

    %
    % accessors for schema instance variables
    %

```

```

    terminal1(Volt1,CurrentIn1).
    terminal2(Volt2,CurrentOut2).
    terminal3(Volt3,CurrentIn3).
    terminal4(Volt4,CurrentOut4).

```

```

    volt1(Volt1). currentIn1(CurrentIn1).
    volt2(Volt2). currentIn2(CurrentOut2).
    volt3(Volt3). currentIn3(CurrentIn3).
    volt4(Volt4). currentIn4(CurrentOut4).

```

```

    %
    % relay methods
    %

```

```

    %
    % initialization - relay has state 0 to 5
    %

```

```

    :- State = {0..5} _ .

```

```

    %
    % mode/0 defines behavioral modes of solenoid
    %

```

```

    order mode.

```

```

    mode:- % relay not energized
           volt1 == Volt2,

```

```

    CurrentIn1 ::= 0,
    CurrentOut2 ::= 0,
    CurrentIn3 ::= 0,
    CurrentOut4 ::= 0,
    State ::= 0,
    Condition ::= 0.

mode:- % relay energized
    Volt1 - Volt2 >= 9,
    Volt1 - Volt2 <= 16,
    Volt3 ::= Volt4,
    CurrentIn1 ::= CurrentOut2, % primary current
    CurrentIn3 ::= CurrentOut4, % secondary current
    State ::= 1,
    Condition ::= 0.

mode:- % open circuit in primary
    CurrentIn1 ::= 0,
    CurrentOut2 ::= 0,
    CurrentIn3 ::= 0,
    CurrentOut4 ::= 0,
    State ::= 2,
    Condition ::= 1.

mode:- % short circuit in primary
    Volt1 ::= Volt2,
    Volt3 ::= Volt4,
    CurrentIn1 ::= CurrentOut2, % primary current
    CurrentIn3 ::= CurrentOut4, % secondary current
    State ::= 3,
    Condition ::= 1.

mode:- % open circuit in secondary
    CurrentIn3 ::= 0,
    CurrentOut4 ::= 0,
    State ::= 4,
    Condition ::= 1.

mode:- % short circuit in secondary
    Volt3 ::= Volt4,
    State ::= 5,
    Condition ::= 1.
}

%-----
%
% Schema tank is derived from primitve. This is a model of a simple
% fuel tank
%
%-----

```

```
schema tank:primitive
```

```
{
    %
    % Schema instance variables
    %

    pressRange Press1. flowRateRange FlowIn1. conditionType FlowCond1.
    pressRange Press2. flowRateRange FlowOut2. conditionType FlowCond2.
    pressRange Press3. flowRateRange FlowOut3. conditionType FlowCond3.

    %
    % accessors for schema instance variables
    %

    terminal1(Press1,FlowIn1,FlowCond1).
    terminal2(Press2,FlowOut2,FlowCond2).
    terminal3(Press3,FlowOut3,FlowCond3).

    press1(Press1). flowIn1(FlowIn1). flowCond1(FlowCond1).
    press2(Press2). flowOut2(FlowOut2). flowCond2(FlowCond2).
    press3(Press3). flowOut3(FlowOut3). flowCond3(FlowCond3).

    %
    % tank methods
    %

    %
    % initialization - tank has state 0 to 1
    %

    :- State = {0..1} _ .

    %
    % mode/0 defines behavioral modes of solenoid
    %

    order mode.

    mode:-Press1 == Press2,
        State == 0,
        Condition == 0.

    mode:-Press1 \= Press2,
        State == 1,
        Condition == 1.
}

%-----
%
% Schema regulator is derived from primitive. This is a model of a simple
% diaphragm pressure regulator
```

```

%
%-----
schema regulator:primitive
{
    %
    % Schema instance variables
    %

    pressRange Press1. flowRateRange FlowIn1. conditionType FlowCond1.
    pressRange Press2. flowRateRange FlowOut2. conditionType FlowCond2.
    pressRange Press3. flowRateRange FlowOut3. conditionType FlowCond3.

    %
    % accessors for schema instance variables
    %

    terminal1(Press1,FlowIn1,FlowCond1).
    terminal2(Press2,FlowOut2,FlowCond2).
    terminal3(Press3,FlowOut3,FlowCond3).

    press1(Press1). flowIn1(FlowIn1). flowCond1(FlowCond1).
    press2(Press2). flowOut2(FlowOut2). flowCond2(FlowCond2).
    press3(Press3). flowOut3(FlowOut3). flowCond3(FlowCond3).

    %
    % regulator methods
    %

    %
    % initialization - regulator has state 0 to 4
    %

    :- State = {0..4} _ .

    %
    % mode/0 defines behavioral modes of solenoid
    %

    order mode.

    mode:- % return flow to tank
           FlowIn1 == FlowOut3,
           Press1 - Press2 > 5,
           State == 0,
           Condition == 0.

    mode:- % no return flow
           FlowIn1 == 0,
           FlowOut3 == 0,
           Press1 - Press2 < 5,

```

```

        State ::= 1,
        Condition ::= 0.

mode:- % faulty diaphragm (will not open)
        FlowIn1 ::= 0,
        FlowOut3 ::= 0,
        State ::= 2,
        Condition ::= 1.

mode:- % faulty diaphragm (will not close)
        FlowIn1 ::= FlowOut3,
        State ::= 3,
        Condition ::= 1.

mode:- % leaking regulator
        FlowIn1 =\= FlowOut3,
        State ::= 4,
        Condition ::= 1.
}

%-----
%
% Schema pump is derived from primitive. This is a model of a simple
% dc pump
%
%-----

schema pump:primitive
{
    %
    % Schema instance variables
    %

    pressRange Press1. flowRateRange FlowIn1. conditionType FlowCond1.
    pressRange Press2. flowRateRange FlowOut2. conditionType FlowCond2.
    voltageRange Volt3. currentRange CurrentIn3.
    voltageRange Volt4. currentRange CurrentOut4.

    %
    % accessors for schema instance variables
    %

    terminal1(Press1,FlowIn1,FlowCond1).
    terminal2(Press2,FlowOut2,FlowCond2).
    terminal3(Volt3,CurrentIn3).
    terminal4(Volt4,CurrentOut4).

    press1(Press1). flowIn1(FlowIn1). flowCond1(FlowCond1).
    press2(Press2). flowOut2(FlowOut2). flowCond2(FlowCond2).
    volt3(Volt3). currentIn3(CurrentIn3).
    volt4(Volt4). currentOut4(CurrentOut4).

```

```

%
% pump methods
%

%
% initialization - pump has state 0 to 4
%

:- State = {0..4} _;

%
% mode/0 defines behavioral modes of solenoid
%

order mode.

mode:- % pump 'on'
    FlowIn1 == FlowOut2,
    Press2 > Press1,
    Volt3 - Volt4 >= 9, % Power - ECM voltage
    Volt3 - Volt4 <= 16,
    CurrentOut4 == CurrentIn3,
    State == 0,
    Condition == 0.

mode:- % pump 'off'
    FlowIn1 == 0,
    FlowOut2 == 0,
    Volt3 == Volt4,
    CurrentIn3 == 0,
    CurrentOut4 == 0,
    State == 1,
    Condition == 0.

mode:- % weak pump
    Volt3 - Volt4 >= 9,
    Volt3 - Volt4 <= 16,
    Press2 <= Press1,
    FlowIn1 == FlowOut2,
    State == 2,
    Condition == 1.

mode:- % open circuited pump
    FlowIn1 == 0,
    FlowOut2 == 0,
    State == 3,
    Condition == 1.

mode:- % leaking pump
    FlowIn1 \= FlowOut2,

```

```

        State ::= 4,
        Condition ::= 1.
    }

%-----
%
% Schema injector is derived from primitive. This is a model of a simple
% fuel injector
%
%-----

schema injector:primitive
{
    %
    % Schema instance variables
    %

    pressRange Press1. flowRateRange FlowOut1. conditionType FlowCond1.
    pressRange Press2. flowRateRange FlowIn2. conditionType FlowCond2.
    voltageRange Volt3. currentRange CurrentIn3.
    voltageRange Volt4. currentRange CurrentOut4.

    %
    % accessors for schema instance variables
    %

    terminal1(Press1,FlowOut1,FlowCond1).
    terminal2(Press2,FlowIn2,FlowCond2).
    terminal3(Volt3,CurrentIn3).
    terminal4(Volt4,CurrentOut4).

    press1(Press1). flowIn1(FlowOut1). flowCond1(FlowCond1).
    press2(Press2). flowOut2(FlowIn2). flowCond2(FlowCond2).
    volt3(Volt3). currentIn3(CurrentIn3).
    volt4(Volt4). currentOut4(CurrentOut4).

    %
    % injector methods
    %

    %
    % initialization - injector has state 0 to 4
    %

    :- State = {0..4} _ .

    %
    % mode/0 defines behavioral modes of solenoid
    %

    order mode.

```



```

mode:- % injector 'on'
      FlowOut1 ::= FlowIn2,
      Volt3 - Volt4 =< 16,
      Volt3 - Volt4 >= 9,
      CurrentIn3 ::= CurrentOut4,
      State ::= 0,
      Condition ::= 0.

mode:- % injector 'off'
      FlowIn2 ::= 0,
      FlowOut1 ::= 0,
      Volt3 ::= Volt4,
      CurrentIn3 ::= 0,
      CurrentOut4 ::= 0,
      State ::= 1,
      Condition ::= 0.

mode:- % injector leaking
      FlowOut1 =\= FlowIn2,
      State ::= 2,
      Condition ::= 1.

mode:- % injector plugged or open circuited
      FlowOut1 ::= 0,
      FlowIn2 ::= 0,
      State ::= 3,
      Condition ::= 1.

mode:- % short circuited
      FlowOut1 ::= FlowIn2,
      State ::= 4,
      Condition ::= 1.
}

%-----
%
% Schema filter is derived from primitve. This is a model of a simple
% filter.
%
%-----

schema filter:primitive
{
  %
  % Schema instance variables
  %

  pressRange Press1. flowRateRange FlowIn1. conditionType FlowCond1.
  pressRange Press2. flowRateRange FlowOut2. conditionType FlowCond2.

```

```

%
% accessors for schema instance variables
%

terminal1(Press1,FlowIn1,FlowCond1).
terminal2(Press2,FlowOut2,FlowCond2).

press1(Press1). flowIn1(FlowIn1). flowCond1(FlowCond1).
press2(Press2). flowOut2(FlowOut2). flowCond2(FlowCond2).

%
% filter methods
%

%
% initialization - filter has state 0 to 2
%

:- State = {0..2} _ .

%
% mode/0 defines behavioral modes of solenoid
%

order mode.

mode:- % good state
      Press2 == Press1,
      FlowOut2 == FlowIn1,
      State == 0,
      Condition == 0.

mode:- % blocked filter
      Press2 \= Press1,
      State == 1,
      Condition == 1.

mode:- % leaking filter
      FlowOut2 \= FlowIn1,
      State == 2,
      Condition == 1.
}

%-----
%
% Schema rail is derived from primitive. This is a model of a simple
% fluid junction with three branches.
%
%-----

schema rail:primitive

```

```

%
% Schema instance variables
%

pressRange Press1. flowRateRange FlowIn1. conditionType FlowCond1.
pressRange Press2. flowRateRange FlowOut2. conditionType FlowCond2.
pressRange Press3. flowRateRange FlowOut3. conditionType FlowCond3.

%
% accessors for schema instance variables
%

terminal1(Press1,FlowIn1,FlowCond1).
terminal2(Press2,FlowOut2,FlowCond2).
terminal3(Press3,FlowOut3,FlowCond3).

press1(Press1). flowIn1(FlowIn1). flowCond1(FlowCond1).
press2(Press2). flowOut2(FlowOut2). flowCond2(FlowCond2).
press3(Press3). flowOut3(FlowOut3). flowCond3(FlowCond3).

% This allows for 1 injector

%
% rail methods
%

%
% initialization - rail has state 0 to 1
%

:- State = {0..1} _ .

%
% mode/0 defines behavioral modes of solenoid
%

order mode.

mode:- % good state
      FlowIn1 == FlowOut2 + FlowOut3,
      Press1 == Press2,
      Press1 == Press3,
      State == 0,
      Condition == 0.

mode:- % leaking rail
      FlowIn1 \= FlowOut2 + FlowOut3,
      State == 1,
      Condition == 1.
}

```

A.3. Compound Knowledge Base

```

%-----
%
% Automotive Diagnosis
% charlie/diagnosis/new/compound.kb
%
% This file contains schemas for compound component models. All inherit
% from schema compound in components.kb.
%
%-----

```

```

%-----
%
% Schema igswitch is derived from compound. This is a model of a
% simplified ignition switch. It uses models of a simple switch and a
% junction to derive the behaviour of the ignition switch.
%
%-----

```

```

schema igswitch:compound
{
    %
    % Schema instance variables
    %

    voltageRange Volt1. currentRange CurrentIn1.
    voltageRange Volt2. currentRange CurrentOut2.
    voltageRange Volt3. currentRange CurrentOut3.
    signalType Signal.

    %
    % sub-components of igswitch
    %

    switch Switch. junction3 Junction.

    %
    % accessors for schema instance variables
    %

    terminal1(Volt1,CurrentIn1).
    terminal2(Volt2,CurrentOut2).
    terminal3(Volt3,CurrentOut3).

    volt1(Volt1). currentIn1(CurrentIn1).

```

```

volt2(Volt2). currentOut2(CurrentOut2).
volt3(Volt3). currentOut3(CurrentOut3).
signal(Signal).

%
% igswitch methods
%

%
% build/0 instantiates sub-components of igswitch, gives each
% a name, defines the component list Clist, unifies the
% instance variables of the sub-components, and sends a
% message to compound to build igswitch's sub-components.
%

%
% define components of system
%

build:- Switch isa switch,    Switch:name(iSswitch),
        Junction isa junction3, Junction:name(iSjunction),

        Clist = [Switch,Junction],

        %
        % unify instance variables of power with those of its
        % sub-components
        %

        Switch:terminal1(Volt1, CurrentIn1),
        Switch:signal(Signal),

        Junction:terminal2(Volt2,currentRange I2), I2 := 0-CurrentOut2,
        Junction:terminal3(Volt3,currentRange I3), I3 := 0-CurrentOut3,

        %
        % unify instance variables of between sub-components
        %

        Switch:terminal2(VoltA,CurrentA),

        Junction:terminal1(VoltA,CurrentA),

        %
        % send message to compound to build igswitchs
        % sub-components
        %

compound::build.

%

```

```
% mode/0 assigns value to Condition  
%
```

```
order mode.
```

```
mode:-Switch:state(0),  
      Junction:state(0),  
      Signal ::= off,  
      Condition ::= 0.
```

```
mode:-Switch:state(1),  
      Junction:state(0),  
      Signal ::= on,  
      Condition ::= 0.
```

```
mode:-Condition ::= 1.
```

```
}
```

A.4. System Knowledge Base

```

%-----
%
% Automotive Diagnosis
% charlie/diagnosis/new/system.kb
%
% This file contains schemas for system models. All system models
% inherit from system in components.kb.
%-----

%-----
%
% Schema power is derived from system. This is a model of a 12-volt
% power distribution system in an automobile. It derives its behaviour
% from that of wires, junctions, fuses, a battery, and an ignition
% switch
%
%-----

schema power:system
{
    %
    % Schema instance variables
    %

    voltageRange Volt1.    currentRange CurrentOut1.
    voltageRange Volt2.    currentRange CurrentOut2.
    voltageRange Volt3.    currentRange CurrentOut3.
    voltageRange Volt4.    currentRange CurrentOut4.
    voltageRange Volt5.    currentRange CurrentOut5.

    signalType Signal.

    %
    % sub-components of power
    %

    igswitch IS.           battery Battery.
    wire W1.                wire W2.           wire W6.
    fuse ECM_F.            fuse Engine_F.
    junction3 J1.          junction3 J2.       junction3 J3.

    %
    % accessors for persistent variables

```

```

%

terminal1(Volt1,CurrentOut1).
terminal2(Volt2,CurrentOut2).
terminal3(Volt3,CurrentOut3).
terminal4(Volt4,CurrentOut4).
terminal5(Volt5,CurrentOut5).

volt1(Volt1).  currentOut1(CurrentOut1).
volt2(Volt2).  currentOut2(CurrentOut2).
volt3(Volt3).  currentOut3(CurrentOut3).
volt4(Volt4).  currentOut4(CurrentOut4).
volt5(Volt5).  currentOut5(CurrentOut5).

igswitch(IS).      batt(Battery).      w3(W6).
w1(W1).            w2(W2).
ecm_f(ECM_F).      engine_f(Engine_F).
j1(J1).            j2(J2).            j3(J3).
signal(Signal).

%
% power methods
%

%
% build/0 instantiates sub-components of power, assigns each
% a name, defines the component list Clist, unifies the
% instance variables of the sub-components, and sends a
% message to system to build power's sub-components.
%

build:-
    IS isa igswitch,          IS:name(ignitionswitch),
    Battery isa battery,     Battery:name(battery),
    W1 isa wire,             W1:name(wire1),
    W2 isa wire,             W2:name(wire2),
    W6 isa wire,             W6:name(wire6),
    J1 isa junction3,       J1:name(junction1),
    J2 isa junction3,       J2:name(junction2),
    J3 isa junction3,       J3:name(junction3),
    ECM_F isa fuse,         ECM_F:name(ecmfuse),
    Engine_F isa fuse,      Engine_F:name(enginefuse),

    Clist = [IS,Battery,W1,W2,W6,J1,J2,J3,ECM_F,Engine_F],

%
% unify instance variables of power with those of its
% sub-components
%

J1:terminal2(Volt1,currentRange I1), 0-CurrentOut1 := I1,

```



```
J1:terminal3(Volt2,currentRange I2), 0-CurrentOut2 ::= I2,
```

```
J2:terminal2(Volt3,currentRange I3), 0-CurrentOut3 ::= I3,
```

```
J2:terminal3(Volt4,currentRange I4), 0-CurrentOut4 ::= I4,
```

```
J3:terminal3(Volt5,currentRange I5), 0-CurrentOut5 ::= I5,
```

```
IS:signal(Signal),
```

```
%
```

```
% unify instance variables of between sub-components
```

```
%
```

```
Battery:terminal(VoltA,CurrentA),
```

```
J3:terminal1(VoltA,CurrentA),
```

```
J3:terminal2(VoltB,currentRange CurrentB),
```

```
W1:terminal1(VoltB,currentRange CurrentBB),
```

```
0-CurrentB ::= CurrentBB,
```

```
W1:terminal2(VoltC,CurrentC),
```

```
IS:terminal1(VoltC,CurrentC),
```

```
IS:terminal2(VoltD,CurrentD),
```

```
W2:terminal1(VoltD,CurrentD),
```

```
IS:terminal3(VoltE,CurrentE),
```

```
W6:terminal1(VoltE,CurrentE),
```

```
W6:terminal2(VoltF,CurrentF),
```

```
ECM_F:terminal1(VoltF,CurrentF),
```

```
W2:terminal2(VoltG,CurrentG),
```

```
Engine_F:terminal1(VoltG,CurrentG),
```

```
Engine_F:terminal2(VoltH,CurrentH),
```

```
J1:terminal1(VoltH,CurrentH),
```

```
ECM_F:terminal2(VoltI,CurrentI),
```

```
J2:terminal1(VoltI,CurrentI),
```

```
%
```

```
% send message to system to build power's
```

```
% sub-components
```

```
%
```

```
system::build.
```

```
%
```

```
%
```

```
% Schema ccp is derived from system. This is a model of a charcoal
% canister purge system.
```

```
%
```

```
%-----
```

```
schema ccp:system
```

```
{
```

```
  %
```

```
  % Schema instance variables
```

```
  %
```

```
  pressRange Press1. flowRateRange FlowOut1. conditionType FlowCond1.
  voltageRange Volt2.          currentRange CurrentIn2.
  voltageRange Volt3.          currentRange CurrentOut3.
  pressRange Press4. flowRateRange FlowIn4. conditionType FlowCond4.
  pressRange Press5. flowRateRange FlowIn5. conditionType FlowCond5.
  pressRange Press6. flowRateRange FlowOut6. conditionType FlowCond6.
```

```
  %
```

```
  % sub-components of ccp
```

```
  %
```

```
  canister Can.          pcv PCV.          solenoid Sol.
  hose H1.              hose H2.          hose H3.
  hose H5.              hose H6.
  wire W1.              wire W2.
```

```
  %
```

```
  % accessors for persistent variables
```

```
  %
```

```
  terminal1(Press1,FlowOut1,FlowCond1).
  terminal2(Volt2,CurrentIn2).
  terminal3(Volt3,CurrentOut3).
  terminal4(Press4,FlowIn4,FlowCond4).
  terminal5(Press5,FlowIn5,FlowCond5).
  terminal6(Press6,FlowOut6,FlowCond6).
```

```
  press1(Press1). flowOut1(FlowOut1). flowCond1(FlowCond1).
  volt2(Volt2). currentIn2(CurrentIn2).
  volt3(Volt3). currentOut3(CurrentOut3).
  press4(Press4). flowIn4(FlowIn4). flowCond4(FlowCond4).
  press5(Press5). flowIn5(FlowIn5). flowCond5(FlowCond5).
  press6(Press6). flowOut6(FlowOut6). flowCond6(FlowCond6).
```

```
  can(Can).          pCV(PCV).          sol(Sol).
  h1(H1).           h2(H2).          h3(H3).
  h5(H5).           h6(H6).
```

```
  %
```

```
  % schema methods
```

```

%

%
% build/0 instantiates sub-components of power, assigns each
% a name, defines the component list Clist, unifies the
% instance variables of the sub-components, and sends a
% message to system to build schema's sub-components.
%

build:-
    Can isa canister,      Can:name(canister1),
    PCV isa pcv,          PCV:name(presCV),
    Sol isa solenoid,     Sol:name(solenoid1),
    H1 isa hose,          H1:name(hose1),
    H2 isa hose,          H2:name(hose2),
    H3 isa hose,          H3:name(hose3),
    H5 isa hose,          H5:name(hose5),
    H6 isa hose,          H6:name(hose6),
    W1 isa wire,          W1:name(wire1),
    W2 isa wire,          W2:name(wire2),

    Clist = [Sol,Can,PCV,H1,H2,H3,H5,H6,W1,W2],

%
% unify instance variables of schema with those of its
% sub-components
%

H1:terminal2(Press1,FlowOut1,FlowCond1),

W1:terminal1(Volt2,CurrentIn2),

W2:terminal2(Volt3,CurrentOut3),

Can:terminal2(Press4,FlowIn4,FlowCond4),

H5:terminal1(Press5,FlowIn5,FlowCond5),

H6:terminal2(Press6,FlowOut6,FlowCond6),

%
% unify instance variables of between sub-components
%

Sol:terminal1(PressA,FlowA,CondA),
H1:terminal1(PressA,FlowA,CondA),

Sol:terminal2(PressB,FlowB,CondB),
H2:terminal2(PressB,FlowB,CondB),

Sol:terminal3(VoltC,CurrentC),

```

```

W1:terminal2(VoltC,CurrentC),

Sol:terminal4(VoltD,CurrentD),
W2:terminal1(VoltD,CurrentD),

Can:terminal1(PressE,FlowE,CondE),
H2:terminal1(PressE,FlowE,CondE),

Can:terminal3(PressF,FlowF,CondF),
H3:terminal2(PressF,FlowF,CondF),

PCV:terminal1(PressG,FlowG,CondG),
H3:terminal1(PressG,FlowG,CondG),

PCV:terminal2(PressH,FlowH,CondH),
H5:terminal2(PressH,FlowH,CondH),

PCV:terminal3(PressI,flowRateRange FlowI,CondI),
H6:terminal1(PressI,flowRateRange FlowIR,CondI),
FlowIR ::= 0 - FlowI,

%
% send message to system to build schema's
% sub-components
%

system::build.
}

%-----
%
% Schema fuel is derived from system. This is a model of an electrically
% controlled fuel injection system.
%
%-----

schema fuel:system
{
    %
    % Schema instance variables
    %

    voltageRange Volt1. currentRange CurrentOut1.
    voltageRange Volt2. currentRange CurrentIn2.
    voltageRange Volt3. currentRange CurrentIn3.
    voltageRange Volt4. currentRange CurrentOut4.
    voltageRange Volt5. currentRange CurrentIn5.
    pressRange Press6. flowRateRange FlowOut6. conditionType FlowCond6.
    pressRange Press7. flowRateRange FlowOut7. conditionType FlowCond7.
    pressRange Press8. flowRateRange FlowOut8. conditionType FlowCond8.
    signalType Signal.

```

```

%
% sub-components of fuel
%

tank F_Tank.      pump F_Pump.      filter F_Filter.  rail Rail.
injector Inj.     regulator Press_Reg.
hose H1.          hose H2.          hose H3.          hose H4.
fuse F_Fuse.     switch F_Switch.
relay Relay.     junction4 J4.     junction3 J3.
wire W1.         wire W2.
wire W3.         wire W4.          wire W5.          wire W6.
wire W7.         wire W8.          wire W9.          wire W10.

```

```

%
% accessors for persistent variables
%

```

```

terminal1(Volt1,CurrentOut1).
terminal2(Volt2,CurrentIn2).
terminal3(Volt3,CurrentIn3).
terminal4(Volt4,CurrentOut4).
terminal5(Volt5,CurrentIn5).
terminal6(Press6,FlowOut6,FlowCond6).
terminal7(Press7,FlowOut7,FlowCond7).
terminal8(Press8,FlowOut8,FlowCond8).
signal(Signal).

```

```

volt1(Volt1).    currentOut1(CurrentOut1).
volt2(Volt2).    currentIn2(CurrentIn2).
volt3(Volt3).    currentIn3(CurrentIn3).
volt4(Volt4).    currentOut4(CurrentOut4).
volt5(Volt5).    currentIn5(CurrentIn5).

```

```

press6(Press6). flowOut6(FlowOut6). flowCond6(FlowCond6).
press7(Press7). flowOut7(FlowOut7). flowCond7(FlowCond7).
press8(Press8). flowOut8(FlowOut8). flowCond8(FlowCond8).

```

```

f_Tank(F_Tank).    f_Pump(F_Pump).    f_Filter(F_Filter).  rail(Rail).
inj(Inj).          press_reg(Press_Reg).
h1(H1).            h2(H2).            h3(H3).              h4(H4).
f_Fuse(F_Fuse).   f_switch(F_Switch). relay(Relay).
j4(J4).           j3(J3)
w1(W1).           w2(W2).            w3(W3).              w4(W4).
w5(W5).           w6(W6).            w7(W7).              w8(W8).
w9(W9).           w10(W10).

```

```

%
% schema methods
%

```

```

%
% build/0 instantiates sub-components of power, assigns each
% a name, defines the component list Clist, unifies the
% instance variables of the sub-components, and sends a
% message to system to build schema's sub-components.
%

build:- F_Tank isa tank,                F_Tank:name(fueltank),
        F_Pump isa pump,                F_Pump:name(fuelpump),
        F_Filter isa filter,           F_Filter:name(fuelfilter),
        F_Fuse isa fuse,               F_Fuse:name(fuelfuse),
        F_Switch isa switch,          F_Switch:name(fuelswitch),
        Relay isa relay,              Relay:name(fuelrelay),
        J4 isa junction4,             J4:name(junct4),
        J3 isa junction3,             J3:name(junct3),
        Rail isa rail,                Rail:name(fuelrail),
        Inj isa injector,             Inj:name(fuelinj),
        Press_Reg isa regulator,      Press_Reg:name(fuelreg),
        H1 isa hose,                  H1:name(hose1),
        H2 isa hose,                  H2:name(hose2),
        H3 isa hose,                  H3:name(hose3),
        H4 isa hose,                  H4:name(hose4),
        W1 isa wire,                  W1:name(wire1),
        W2 isa wire,                  W2:name(wire2),
        W3 isa wire,                  W3:name(wire3),
        W4 isa wire,                  W4:name(wire4),
        W5 isa wire,                  W5:name(wire5),
        W6 isa wire,                  W6:name(wire6),
        W7 isa wire,                  W7:name(wire7),
        W8 isa wire,                  W8:name(wire8),
        W9 isa wire,                  W9:name(wire9),
        W10 isa wire,                 W10:name(wire10),

Clist =
[H1,H2,H3,H4,W1,W2,W3,W4,W5,W6,W7,W8,W9,W10,J4,J3,
  Relay,Rail,Inj,Press_Reg,F_Switch,F_Tank,F_Pump,
  F_Filter,F_Fuse],

%
% unify instance variables of schema with those of its
% sub-components
%

W1:terminal2(Volt1,CurrentOut1),

W2:terminal1(Volt2,CurrentIn2),

W3:terminal1(Volt3,CurrentIn3),

W4:terminal2(Volt4,CurrentOut4),

```

```

W5:terminal1(Volt5,CurrentIn5),

Inj:terminal1(Press6,FlowOut6,FlowCond6),

H4:terminal2(Press7,FlowOut7,FlowCond7),

F_Tank:terminal3(Press8,FlowOut8,FlowCond8),

F_Switch:signal(Signal),

%
% unify instance variables of between sub-components
%

F_Switch:terminal1(VoltA,CurrentA),
W10:terminal2(VoltA,CurrentA),

F_Switch:terminal2(VoltB,CurrentB),
W6:terminal1(VoltB,CurrentB),

J3:terminal1(VoltC,currentRange CurrentC1),
W8:terminal1(VoltC,currentRange CurrentC2),
CurrentC1 ::= 0 - CurrentC2,

J3:terminal2(VoltD,currentRange CurrentD1),
W10:terminal1(VoltD,currentRange CurrentD2),
CurrentD1 ::= 0 - CurrentD2,

J4:terminal1(VoltE,CurrentE),
W6:terminal2(VoltE,CurrentE), % from switch

J4:terminal2(VoltF,CurrentF),
W7:terminal2(VoltF,CurrentF), % from relay

J4:terminal3(VoltG,currentRange CurrentG1),
W9:terminal1(VoltG,currentRange CurrentG2), % to pump
CurrentG1 ::= 0 - CurrentG2,

J4:terminal4(VoltH,currentRange CurrentH1),
W1:terminal1(VoltH,currentRange CurrentH2), % to pump
CurrentH1 ::= 0 - CurrentH2,

J4:terminal4(VoltI,CurrentI),
W1:terminal1(VoltI,CurrentI), % to ecm

Relay:terminal1(VoltJ,CurrentJ),
W2:terminal2(VoltJ,CurrentJ),

Relay:terminal3(VoltL,CurrentL),
W8:terminal2(VoltL,CurrentL),

```

Relay:terminal4(VoltM,CurrentM),
W7:terminal1(VoltM,CurrentM),

F_Fuse:terminal1(VoltN,CurrentN),
W3:terminal2(VoltN,CurrentN),

F_Fuse:terminal2(VoltO,CurrentO),
J3:terminal3(VoltO,CurrentO),

Inj:terminal3(VoltP,CurrentP),
W5:terminal2(VoltP,CurrentP),

Inj:terminal4(VoltQ,CurrentQ),
W4:terminal1(VoltQ,CurrentQ),

F_Pump:terminal3(VoltR,CurrentR),
W9:terminal2(VoltR,CurrentR),

F_Tank:terminal2(PressureA,FlowA,FCondA),
F_Pump:terminal1(PressureA,FlowA,FCondA),

F_Tank:terminal1(PressureB,FlowB,FCondB),
H3:terminal2(PressureB,FlowB,FCondB),

F_Pump:terminal1(PressureC,FlowC,FCondC),
F_Tank:terminal2(PressureC,FlowC,FCondC),

F_Pump:terminal2(PressureD,FlowD,FCondD),
H1:terminal1(PressureD,FlowD,FCondD),

F_Filter:terminal1(PressureE,FlowE,FCondE),
H1:terminal2(PressureE,FlowE,FCondE),

F_Filter:terminal2(PressureF,FlowF,FCondF),
H2:terminal1(PressureF,FlowF,FCondF),

Rail:terminal1(PressureG,FlowG,FCondG),
H2:terminal2(PressureG,FlowG,FCondG),

Rail:terminal2(PressureH,FlowH,FCondH),
Inj:terminal1(PressureH,FlowH,FCondH),

Rail:terminal3(PressureI,FlowI,FCondI),
Press_Reg:terminal1(PressureI,FlowI,FCondI),

Press_Reg:terminal2(PressureJ,FlowJ,FCondJ),
H4:terminal1(PressureJ,FlowJ,FCondJ),

Press_Reg:terminal3(PressureK,FlowK,FCondK),
H3:terminal1(PressureK,FlowK,FCondK),


```

%
% send message to system to build schema's
% sub-components
%

    system::build.
)

%-----
%
% Schema engine is derived from system.
%
%-----

schema engine:system
{
    %
    % Schema instance variables
    %

    voltageRange Volt1.    currentRange CurrentIn1.
    voltageRange Volt2.    currentRange CurrentIn2.
    voltageRange Volt3.    currentRange CurrentIn3.
    pressRange Press4. flowRateRange FlowIn4. conditionType FlowCond4.

    %
    % sub-components of engine
    %

    fuel F.
    ccp C.

    %
    % accessors for persistent variables
    %

    terminal1(Volt1,CurrentIn1).
    terminal2(Volt2,CurrentIn2).
    terminal3(Volt3,CurrentIn3).
    terminal4(Press4,FlowIn4,FlowCond4).

    volt1(Volt1).    currentIn1(CurrentIn1).
    volt2(Volt2).    currentIn2(CurrentIn2).
    volt3(Volt3).    currentIn3(CurrentIn3).
    press1(Press4). flowIn1(FlowIn4). flowCond4(FlowCond4).

    f(F).
    c(C).

    %
    % schema methods

```

```

%

%
% build/0 instantiates sub-components of power, assigns each
% a name, defines the component list Clist, unifies the
% instance variables of the sub-components, and sends a
% message to system to build schema's sub-components.
%

build:- F isa fuel, F:name(fuelSyst),
        C isa ccp, C:name(ccpSyst),

        Clist = [F,C],

        %
        % unify instance variables of schema with those of its
        % sub-components
        %

        F:terminal3(Volt1,CurrentIn1),

        F:terminal5(Volt2,CurrentIn2),

        C:terminal2(Volt3,CurrentIn3),

        C:terminal4(Press4,FlowIn4,FlowCond4),

        %
        % unify instance variables of between sub-components
        %

        F:terminal8(PressA,FlowA,CondA),
        C:terminal5(PressA,FlowA,CondA),

        %
        % send message to system to build schema's
        % sub-components
        %

        system::build.
}

%-----
%
% Schema vehicle is derived from system.
%
%-----

schema vehicle:system
{
    %

```

```

% Schema instance variables
%

signalType Signal.
pressRange Press. flowRateRange FlowIn. conditionType FlowCond.

%
% sub-components of vehicle
%

power P.
engine E.

%
% accessors for persistent variables
%

signal(Signal).
terminal(Press,FlowIn,FlowCond).
press(Press). flowIn(FlowIn). flowCond(FlowCond).

p(P).
e(E).

%
% schema methods
%

%
% build/0 instantiates sub-components of power, assigns each
% a name, defines the component list Clist, unifies the
% instance variables of the sub-components, and sends a
% message to system to build schema's sub-components.
%

build:- P isa power, P:name(powerSyst),

      E isa engine, E:name(engineSyst),

      Clist = [P,E],

      %
      % unify instance variables of schema with those of its
      % sub-components
      %

      P:signal(Signal),

      E:terminal4(Press,FlowIn,FlowCond),

      %

```

```
% unify instance variables of between sub-components  
%
```

```
P:terminal2(VoltA,CurrentA),  
E:terminal3(VoltA,CurrentA),
```

```
P:terminal4(VoltB,CurrentB),  
E:terminal2(VoltB,CurrentB),
```

```
P:terminal5(VoltC,CurrentC),  
E:terminal1(VoltC,CurrentC),
```

```
%  
% send message to system to build schema's  
% sub-components  
%
```

```
system::build.
```

```
]
```

Appendix B - Simulated Diagnostic Sessions

B.1. Power System Diagnostic Simulation

B.1.1. Input Data-Base File

```
%-----  
%  
% Automotive Diagnosis  
% charlie/diagnosis/new/power1  
%  
% This is a data-base file for the top level commands for a power  
% system diagnostic session.  
%  
%-----  
  
set warnings off  
  
%  
% load knowledge bases  
%  
  
load component.kb  
load primitive.kb  
load compound.kb  
load system.kb  
  
%  
% define, name, and build top level object  
%  
  
power P isa power,P:name(powerSystem).  
P:build.  
  
%  
% find and display a healthy candidate (i.e., no measurements have  
% yet been taken)  
%  
  
P:findCandidates.
```

```
%  
% discard that candidate, add new data, and find new candidates  
%
```

```
undo 2  
P:signal(off),P:volt5(6).  
P:findCandidates.
```

```
%  
% add new data, and find new candidates  
%
```

```
undo 4  
P:batt(battery Battery),Battery:volt(14).  
P:findCandidates.
```

```
%  
% add new data, and find new candidates  
%
```

```
undo 6  
P:j3(junction3 J3),J3:volt2(14).  
P:findCandidates.
```

B.1.2. Output File

Echidna Version 0.9 beta
 Compiled: Thu Oct 10 15:22:59 PDT 1991
 (c) Copyright Expert Systems Lab.
 Simon Fraser University, 1991
 All rights reserved
 (Expires: 8-Mar-92)

```
echidna command 1> load power1
loading data base file "power1" ...
loading knowledge base file "component.kb" ...
loading knowledge base file "primitive.kb" ...
loading knowledge base file "compound.kb" ...
loading knowledge base file "system.kb" ...
query #0 "power P isa power,P:name(powerSystem)." issued
done #0 P isa power, P:name(powerSystem).
query #1 "P:build." issued
done #0 P isa power, P:name(powerSystem).
done #1 P:build.
query #2 "P:findCandidates." issued
```

\$

```
SYSTEM powerSystem
  enginefuse state 0 condition 0
  ecmfuse state 0 condition 0
  junction3 state 0 condition 0
  junction2 state 0 condition 0
  junction1 state 0 condition 0
  wire6 state 0 condition 0
  wire2 state 0 condition 0
  wire1 state 0 condition 0
  battery state 0 condition 0
  ignitionswitch state [[iSjunction, 0], [iSwitch, 0]] condition 0
```

no symptoms

\$

```
done #0 P isa power, P:name(powerSystem).
done #1 P:build.
done #2 P:findCandidates.
query #2 "P:findCandidates." is undone
done #0 P isa power, P:name(powerSystem).
done #1 P:build.
query #3 "P:signal(off),P:volt5(6)." issued
done #0 P isa power, P:name(powerSystem).
```

done #1 P:build.
done #3 P:signal(off), P:volt5(6).
query #4 "P:findCandidates." issued

\$

SYSTEM powerSystem
enginefuse state 0 condition 0
ecmfuse state 0 condition 0
junction3 state 1 condition 1
junction2 state 0 condition 0
junction1 state 0 condition 0
wire6 state 0 condition 0
wire2 state 0 condition 0
wire1 state 0 condition 0
battery state 0 condition 0
ignitionswitch state [[iSjunction, 0], [iSswitch, 0]] condition 0

looking for more single fault candidates

\$

SYSTEM powerSystem
enginefuse state 0 condition 0
ecmfuse state 0 condition 0
junction3 state 3 condition 1
junction2 state 0 condition 0
junction1 state 0 condition 0
wire6 state 0 condition 0
wire2 state 0 condition 0
wire1 state 0 condition 0
battery state 0 condition 0
ignitionswitch state [[iSjunction, 0], [iSswitch, 0]] condition 0

looking for more single fault candidates

\$

SYSTEM powerSystem
enginefuse state 0 condition 0
ecmfuse state 0 condition 0
junction3 state 0 condition 0
junction2 state 0 condition 0
junction1 state 0 condition 0
wire6 state 0 condition 0
wire2 state 0 condition 0
wire1 state 0 condition 0
battery state 1 condition 1
ignitionswitch state [[iSjunction, 0], [iSswitch, 0]] condition 0

looking for more single fault candidates

no more single fault candidates

\$

done #0 P isa power, P:name(powerSystem).
 done #1 P:build.
 done #3 P:signal(off), P:volt5(6).
 done #4 P:findCandidates.
 query #4 "P:findCandidates." is undone
 done #0 P isa power, P:name(powerSystem).
 done #1 P:build.
 done #3 P:signal(off), P:volt5(6).
 query #5 "P:batt(battery Battery),Battery:volt(14)." issued
 done #0 P isa power, P:name(powerSystem).
 done #1 P:build.
 done #3 P:signal(off), P:volt5(6).
 done #5 P:batt(Battery), Battery:volt(14).
 query #6 "P:findCandidates." issued

\$

SYSTEM powerSystem

enginefuse state 0 condition 0
 ecmfuse state 0 condition 0
 junction3 state 1 condition 1
 junction2 state 0 condition 0
 junction1 state 0 condition 0
 wire6 state 0 condition 0
 wire2 state 0 condition 0
 wire1 state 0 condition 0
 battery state 0 condition 0
 ignitionswitch state [[iSjunction, 0], [iSswitch, 0]] condition 0

looking for more single fault candidates

\$

SYSTEM powerSystem

enginefuse state 0 condition 0
 ecmfuse state 0 condition 0
 junction3 state 3 condition 1
 junction2 state 0 condition 0
 junction1 state 0 condition 0
 wire6 state 0 condition 0
 wire2 state 0 condition 0
 wire1 state 0 condition 0
 battery state 0 condition 0
 ignitionswitch state [[iSjunction, 0], [iSswitch, 0]] condition 0

looking for more single fault candidates

no more single fault candidates

\$

done #0 P isa power, P:name(powerSystem).
done #1 P:build.
done #3 P:signal(off), P:volt5(6).
done #5 P:batt(Battery), Battery:volt(14).
done #6 P:findCandidates.
query #6 "P:findCandidates." is undone
done #0 P isa power, P:name(powerSystem).
done #1 P:build.
done #3 P:signal(off), P:volt5(6).
done #5 P:batt(Battery), Battery:volt(14).
query #7 "P:j3(junction3 J3),J3:volt2(14)." issued
done #0 P isa power, P:name(powerSystem).
done #1 P:build.
done #3 P:signal(off), P:volt5(6).
done #5 P:batt(Battery), Battery:volt(14).
done #7 P:j3(J3), J3:volt2(14).
query #8 "P:findCandidates." issued

\$

SYSTEM powerSystem
enginefuse state 0 condition 0
ecmfuse state 0 condition 0
junction3 state 3 condition 1
junction2 state 0 condition 0
junction1 state 0 condition 0
wire6 state 0 condition 0
wire2 state 0 condition 0
wire1 state 0 condition 0
battery state 0 condition 0
ignitionswitch state [[iSjunction, 0], [iSswitch, 0]] condition 0

looking for more single fault candidates

no more single fault candidates

\$

done #0 P isa power, P:name(powerSystem).
done #1 P:build.
done #3 P:signal(off), P:volt5(6).
done #5 P:batt(Battery), Battery:volt(14).
done #7 P:j3(J3), J3:volt2(14).
done #8 P:findCandidates.
echidna command 2>

B.2. Canister Purge System Diagnostic Simulation

B.2.1. Input Data-Base File

```
%-----  
%  
% Automotive Diagnosis  
% charlie/diagnosis/new/ccp1  
%  
% This is a data-base file for the top level commands for a ccp  
% system diagnostic session.  
%  
%-----  
  
set warnings off  
  
%  
% load knowledge bases  
%  
  
load component.kb  
load primitive.kb  
load compound.kb  
load system.kb  
  
%  
% define, name, and build top level object  
%  
  
ccp C isa ccp, C:name(ccpSystem).  
C:build.  
  
%  
% find and display a healthy candidate (i.e., no measurements have  
% yet be taken)  
%  
  
C:findCandidates.  
  
%  
% discard that candidate, add new data, and find new candidates  
%
```

undo 2

C:volt2(14),C:volt3(0).

C:findCandidates.

%

% add new data, and find new candidates

%

undo 4

C:press1(-1),C:flowOut1(2).

C:findCandidates.

%

% add new data, and find new candidates

%

undo 6

C:press6(-1).

C:findCandidates.

%

% add new data, and find new candidates

%

undo 8

C:press5(2),C:flowIn5(2).

C:findCandidates.

B.2.2. Output File

echidna

Echidna Version 0.9 beta
 Compiled: Thu Oct 10 15:22:59 PDT 1991
 (c) Copyright Expert Systems Lab.
 Simon Fraser University, 1991
 All rights reserved
 (Expires: 8-Mar-92)

echidna command 1> load ccp1
 loading data base file "ccp1" ...
 loading knowledge base file "component.kb" ...
 loading knowledge base file "primitive.kb" ...
 loading knowledge base file "compound.kb" ...
 loading knowledge base file "system.kb" ...
 query #0 "C isa ccp, C:name(ccpSystem)." issued
 done #0 C isa ccp, C:name(ccpSystem).
 query #1 "C:build." issued
 done #0 C isa ccp, C:name(ccpSystem).
 done #1 C:build.
 query #2 "C:findCandidates." issued

\$

SYSTEM ccpSystem

wire2 state 0 condition 0
 wire1 state 0 condition 0
 hose6 state 0 condition 0
 hose5 state 0 condition 0
 hose3 state 0 condition 0
 hose2 state 0 condition 0
 hose1 state 0 condition 0
 presCV state 0 condition 0
 canister1 state 0 condition 0
 solenoid1 state 0 condition 0

no symptoms

\$

done #0 C isa ccp, C:name(ccpSystem).
 done #1 C:build.
 done #2 C:findCandidates.
 query #2 "C:findCandidates." is undone
 done #0 C isa ccp, C:name(ccpSystem).
 done #1 C:build.
 query #3 "C:volt2(14),C:volt3(0)." issued

done #0 C isa ccp, C:name(ccpSystem).
done #1 C:build.
done #3 C:volt2(14), C:volt3(0).
query #4 "C:findCandidates." issued

\$

SYSTEM ccpSystem
wire2 state 0 condition 0
wire1 state 0 condition 0
hose6 state 0 condition 0
hose5 state 0 condition 0
hose3 state 0 condition 0
hose2 state 0 condition 0
hose1 state 0 condition 0
presCV state 0 condition 0
canister1 state 0 condition 0
solenoid1 state 0 condition 0

no symptoms

\$

done #0 C isa ccp, C:name(ccpSystem).
done #1 C:build.
done #3 C:volt2(14), C:volt3(0).
done #4 C:findCandidates.
query #4 "C:findCandidates." is undone
done #0 C isa ccp, C:name(ccpSystem).
done #1 C:build.
done #3 C:volt2(14), C:volt3(0).
query #5 "C:press1(-1),C:flowOut1(2)." issued
done #0 C isa ccp, C:name(ccpSystem).
done #1 C:build.
done #3 C:volt2(14), C:volt3(0).
done #5 C:press1(-1), C:flowOut1(2).
query #6 "C:findCandidates." issued

\$

SYSTEM ccpSystem
wire2 state 0 condition 0
wire1 state 0 condition 0
hose6 state 0 condition 0
hose5 state 0 condition 0
hose3 state 0 condition 0
hose2 state 0 condition 0
hose1 state 1 condition 1
presCV state 0 condition 0
canister1 state 0 condition 0
solenoid1 state 0 condition 0

looking for more single fault candidates

\$

SYSTEM ccpSystem
wire2 state 2 condition 1
wire1 state 0 condition 0
hose6 state 0 condition 0
hose5 state 0 condition 0
hose3 state 0 condition 0
hose2 state 0 condition 0
hose1 state 0 condition 0
presCV state 0 condition 0
canister1 state 2 condition 0
solenoid1 state 1 condition 0

looking for more single fault candidates

\$

SYSTEM ccpSystem
wire2 state 0 condition 0
wire1 state 2 condition 1
hose6 state 0 condition 0
hose5 state 0 condition 0
hose3 state 0 condition 0
hose2 state 0 condition 0
hose1 state 0 condition 0
presCV state 0 condition 0
canister1 state 2 condition 0
solenoid1 state 1 condition 0

looking for more single fault candidates

\$

SYSTEM ccpSystem
wire2 state 0 condition 0
wire1 state 0 condition 0
hose6 state 0 condition 0
hose5 state 0 condition 0
hose3 state 0 condition 0
hose2 state 0 condition 0
hose1 state 0 condition 0
presCV state 0 condition 0
canister1 state 2 condition 0
solenoid1 state 3 condition 1

looking for more single fault candidates

\$

SYSTEM ccpSystem

wire2 state 0 condition 0
 wire1 state 0 condition 0
 hose6 state 0 condition 0
 hose5 state 0 condition 0
 hose3 state 0 condition 0
 hose2 state 0 condition 0
 hose1 state 0 condition 0
 presCV state 0 condition 0
 canister1 state 2 condition 0
 solenoid1 state 4 condition 1

looking for more single fault candidates

\$

SYSTEM ccpSystem

wire2 state 0 condition 0
 wire1 state 0 condition 0
 hose6 state 0 condition 0
 hose5 state 0 condition 0
 hose3 state 0 condition 0
 hose2 state 0 condition 0
 hose1 state 0 condition 0
 presCV state 0 condition 0
 canister1 state 0 condition 0
 solenoid1 state 5 condition 1

looking for more single fault candidates

no more single fault candidates

\$

done #0 C isa ccp, C:name(ccpSystem).
 done #1 C:build.
 done #3 C:volt2(14), C:volt3(0).
 done #5 C:press1(-1), C:flowOut1(2).
 done #6 C:findCandidates.
 query #6 "C:findCandidates." is undone
 done #0 C isa ccp, C:name(ccpSystem).
 done #1 C:build.
 done #3 C:volt2(14), C:volt3(0).
 done #5 C:press1(-1), C:flowOut1(2).
 query #7 "C:press6(-1)." issued
 done #0 C isa ccp, C:name(ccpSystem).
 done #1 C:build.
 done #3 C:volt2(14), C:volt3(0).
 done #5 C:press1(-1), C:flowOut1(2).

done #7 C:press6(-1).
query #8 "C:findCandidates." issued

\$

SYSTEM ccpSystem
wire2 state 0 condition 0
wire1 state 0 condition 0
hose6 state 0 condition 0
hose5 state 0 condition 0
hose3 state 0 condition 0
hose2 state 0 condition 0
hose1 state 1 condition 1
presCV state 0 condition 0
canister1 state 0 condition 0
solenoid1 state 0 condition 0

looking for more single fault candidates

\$

SYSTEM ccpSystem
wire2 state 2 condition 1
wire1 state 0 condition 0
hose6 state 0 condition 0
hose5 state 0 condition 0
hose3 state 0 condition 0
hose2 state 0 condition 0
hose1 state 0 condition 0
presCV state 0 condition 0
canister1 state 2 condition 0
solenoid1 state 1 condition 0

looking for more single fault candidates

\$

SYSTEM ccpSystem
wire2 state 0 condition 0
wire1 state 2 condition 1
hose6 state 0 condition 0
hose5 state 0 condition 0
hose3 state 0 condition 0
hose2 state 0 condition 0
hose1 state 0 condition 0
presCV state 0 condition 0
canister1 state 2 condition 0
solenoid1 state 1 condition 0

looking for more single fault candidates

done #0 C isa ccp, C:name(ccpSystem).
done #1 C:build.
done #3 C:volt2(14), C:volt3(0).
done #5 C:press1(-1), C:flowOut1(2).
done #7 C:press6(-1).
done #8 C:findCandidates.
query #8 "C:findCandidates." is undone
done #0 C isa ccp, C:name(ccpSystem).
done #1 C:build.
done #3 C:volt2(14), C:volt3(0).
done #5 C:press1(-1), C:fiowOut1(2).
done #7 C:press6(-1).
query #9 "C:press5(2),C:flowIn5(2)." issued
done #0 C isa ccp, C:name(ccpSystem).
done #1 C:build.
done #3 C:volt2(14), C:volt3(0).
done #5 C:press1(-1), C:flowOut1(2).
done #7 C:press6(-1).
done #9 C:press5(2), C:flowIn5(2).
query #10 "C:findCandidates." issued

\$

SYSTEM ccpSystem

wire2 state 0 condition 0
wire1 state 0 condition 0
hose6 state 0 condition 0
hose5 state 0 condition 0
hose3 state 0 condition 0
hose2 state 0 condition 0
hose1 state 1 condition 1
presCV state 2 condition 0
canister1 state 1 condition 0
solenoid1 state 0 condition 0

looking for more single fault candidates

\$

SYSTEM ccpSystem

wire2 state 0 condition 0
wire1 state 0 condition 0
hose6 state 0 condition 0
hose5 state 0 condition 0
hose3 state 0 condition 0
hose2 state 0 condition 0
hose1 state 0 condition 0
presCV state 2 condition 0
canister1 state 1 condition 0
solenoid1 state 5 condition 1

looking for more single fault candidates

no more single fault candidates

\$

done #0 C isa ccp, C:name(ccpSystem).

done #1 C:build.

done #3 C:volt2(14), C:volt3(0).

done #5 C:press1(-1), C:flowOut1(2).

done #7 C:press6(-1).

done #9 C:press5(2), C:flowIn5(2).

done #10 C:findCandidates.

echidna command 2> quit

References

- Abu-Hanna, A. and Gold, Y. (1988) An Integrated, Deep-Shallow Expert System For Multi-Level Diagnosis of Dynamic Systems. In *Artificial Intelligence in Engineering: Diagnosis and Learning*, edited by J.S. Gero, Elsevier, pp.75-94.
- Ben-Bassat, M., Carlson, R.W., Puri, V.K., Davenport, M.D., Schriver, M.L., Smith, R., Portigal, L.D., Lipnick, E.H. and Weil, M.H. (1980) Pattern Based Interactive Diagnosis of Multiple Disorders: The MEDAS System, *IEEE Transactions in Pattern Anal. Mach. Intell.*, vol. 2, no. 2, pp.148-160.
- Blumberg, P.N., Lavoie, G.A. and Tabaczynski, R.J. (1979) Phenomenological Models for Reciprocating Internal Combustion Engines, *Prog. Energy Combust. Sci.*, Vol. 5, pp.123-167.
- Bratko, I. (1986) *Prolog Programming for Artificial Intelligence*, Addison-Wesley, Don Mills, Ontario.
- Buchanan, B.G. and Shortliffe, E.H. (1984) *Rule Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, Addison-Wesley.
- Cho, D. and Hedrick, J.K. (1989) Automotive Powertrain Modeling for Control, *Transactions of the ASME*, Vol. 111, pp. 568-576.
- Dague, P., Deves, P., Raimon, O. (1987) Troubleshooting: When Modelling is the Trouble, *Proceedings AAAI-87*, pp.600-610.
- Davis, R. and Hamscher, W. (1988) Model Based Reasoning: Troubleshooting. In *Exploring Artificial Intelligence*, edited by H.E. Shrobe and the American Association for Artificial Intelligence. Morgan Kaufman, pp.239-296.

- Davis, R. (1984) Diagnostic Reasoning Based on Structure and Behavior. *Artificial Intelligence* 24, pp.347-410.
- Dobner, D.J. (1983) Dynamic Engine Models for Control Development - Part 1: Non-linear and Linear Model Formulation, *International Journal of Vehicle Design*, special publication SP4, Inderscience Enterprises Ltd., U.K.
- Dobner, D.J. (1980) A Mathematical Engine Model for Development of Dynamic Engine Control, SAE 800054, Detroit, MI.
- Doyle, J. (1979) A Truth Maintenance System. *Artificial Intelligence* 12, pp.127-272.
- Duda, R., Hart, P., Konolige, K. and Reboh, R. (1979) A computer based consultant for mineral exploration. SRI International.
- Finin, T. and Morris, G. (1989) Abductive Reasoning in Multiple Fault Diagnosis. *Artificial Intelligence Review*, 3, pp.129-158.
- Fink, P.K., Lusth, J.C. and Duran, J.W. (1985) A General Expert System Design for Diagnostic Problem Solving, *IEEE Transactions in Pattern Anal. Mach. Intell.*, vol. 7, no. 5, pp.553-560.
- Foss, A.M., Heath, R.P.G., Heyworth, P., Cook, J.a. and McLean, J. (1989) Thermodynamic Simulation of a Turbocharged Spark Ignition Engine for Electronic Control Development, C391/044 IMechE.
- G2. (1988) Trademark of Gensym Corp., 125 Cambridge Park Drive, Cambridge, MA, 02140.
- Genesereth, M.R. (1984) The Use of Design Descriptions in Automated Diagnosis, *Artificial Intelligence* 24, pp.411 - 436.
- Genesereth, M.R. (1981) Diagnosis using Hierarchical Design Models. *Proceedings AAAI-81*, pp.278-283.

- Goebel, R. (1990) A Quick Review of Hypothetical Reasoning based on Abduction. AAAI Spring Symposium on Automated Abduction, Stanford University, Stanford, CA.
- Hamscher, W. (1990a) XDE: Diagnosing Devices with Hierarchic Structure and Known Component Failure Modes. IEEE Conference on AI Applications.
- Hamscher, W. (1990b) Modelling Digital Circuits for Troubleshooting: An Overview. IEEE Conference on AI Applications.
- Hamscher, W. and Davis, R. (1987) Issues in Model Based Troubleshooting. Memo 893, MIT Artificial Intelligence Laboratory.
- Hayes-Roth, F., Waterman, D.,A., Lenat, D.B. (1983) *Building Expert Systems*, Addison-Wesley.
- Howe, E., Cohen, P., Dixon, J., Simmons, M. (1986) Dominic: A Domain-Independent Program for Mechanical Engineering Design, *Artificial Intelligence*, Vol.1, No.1, pp. 23-28.
- Isermann, R (1984) Process Fault Detection Based on Modeling and Estimation Methods - A Survey, *Automatica*, Vol. 20, No. 4, pp. 387-404.
- Joseph, S.J. and McCarney, J (1989) a new Engine Analysis System for Sensor and Actuator Related problems, SAE 891726, Detroit, MI.
- KEE (1986)Trademark of Intellicorp, 1975 El Camino Real West, Mountain View, Ca 904040.
- Klausmeier, R. (1986) Using Artificial Intelligence in Vehicle Diagnostic Systems, SAE 861124, Detroit, MI.
- de Kleer, J. and Williams, B.C. (1989) Diagnosis with Behavioral Modes.*Proceedings IJCAI-89*, Detroit, MI, pp.1324-1330.

- de Kleer, J. and Williams, B.C. (1987) Diagnosing Multiple Faults. *Artificial Intelligence* 32, pp.97-130.
- de Kleer, J. (1986a) An Assumption-Based Truth Maintenance System. *Artificial Intelligence* 28, pp.127-162.
- de Kleer, J. (1986b) Extending the ATMS. *Artificial Intelligence* 28, pp.163-196.
- Lee, M.H., Hunt, J.E., Price, C.J. and Long, F.W. (1990) REPAIR: A Model-Based Diagnosis System, UK IT 1990 Conference, Southampton, UK.
- Luger, G.F., Stubblefield, W.A. (1989) *Artificial Intelligence and the Design of Expert Systems*, Benjamin/Cummings.
- Martin, W.A., and Fateman, R.J. (1971) The MACSYMA system. *Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation*, Los Angeles, pp.59-75.
- Min, P.S. and Ribbens, B.R. (1989) A Vector Space Solution to Incipient Sensor Failure Detection with Applications to Automotive Environments. *IEEE Transactions on Vehicular Technology*, Vol. 38, No. 3, pp.148-158.
- Mittal, S., Dym, C.L. and Morjaria, M. (1986) PRIDE: An Expert System For The Design of Paper Handling Systems. *Computer*, July, pp.102-114.
- Morel, T., Keribar, R. and Blumberg, P. (1988) A New Approach to Integrating Engine Performance and Component Design Analysis, SAE 880103, Detroit, MI.
- Moskwa, J.J., and Hedrick, J.K. (1987) Automotive Engine Modeling for Real Time Control Application, *Proc. of American Control Conference*.
- Papoulis, A. (1984) *Probability, Random Variables, and Stochastic Processes*, McGraw Hill, USA.

- Pearl, J. (1988) *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, Morgan Kaufman, San Mateo, CA.
- Peng, Y. and Reggia, J.A. (1986) Plausibility of Diagnostic Hypotheses: The Nature of Simplicity. *Proceedings AAAI-86*, Philadelphia, PA, pp.140-145.
- Powell, J.D. (1987) A Review of IC Engine Models for Control System Design, IFAC 10th Triennial World Congress, Munich.
- Reiter, R. (1987) A Theory of Diagnosis from First Principles. *Artificial Intelligence* 32, pp.57-95.
- Rizzoni, G., Hampo, R., Liubakka, M. and Marko, K. (1989) Real-Time Detection Filters for On-Board Diagnosis of Incipient Failures, SAE 890763, Detroit, MI.
- Shannon, C.E., and Weaver, W. (1949) *The Mathematical Theory of Communication*, University of Illinois Press, Urbana.
- Shirley, M. and Davis R. (1983) Generating Distinguishing Tests based on Hierarchical Models and Symptom Information. *Proceedings IEEE International Conference on Computer Design*, Rye, NY, pp.455-458.
- Sidebottom, S., Havens, W., Cuperman, M., Davison, R., Sidebottom, G. (1991) Echidna Constraint Reasoning System (Version1): Programming Language Manual, technical report in preparation, Center for Systems Science, Simon Fraser University, B.C., Canada.
- Sterling L. and Shapiro, E. (1986) *The Art of Prolog: Advanced Programming Techniques*, MIT Press, Cambridge, MA.
- Struss, P. and Dressler, O. (1989) "Physical Negation"- Integrating Fault Modes into the General Diagnosis Engine. *Proceedings JCAI-89*, Detroit, MI, pp.1318-1323.

Struss, P. (1988) Extensions to ATMS Based Diagnosis. In *Artificial Intelligence in Engineering: Diagnosis and Learning*, edited by J.S. Gero, Elsevier, pp.3-28.

Tomikasi, T., Kishi, N., Hidetoshi, K. and Hino, A. (1987) Application of an Expert System to Engine Troubleshooting, SAE 870910, Detroit, M.I.

Willsky, A.S. (1976) A Survey of Design Methods for Failure Detection in Dynamic Systems, *Automatica*, Vol. 12, pp. 601-611.