UAH Research Report No. ME-90-101
Date of Issue: July 1990

ENGINE DATA INTERPRETATION SYSTEM (EDIS)

Prepared by:

Thomas L. Cost and Martin O. Hofmann
College of Engineering
The University of Alabama in Huntsville
Huntsville, AL 35899

Prepared for:

George C. Marshall Space Flight Center
National Aeronautics and Space Administration
Marshall Space Flight Center, AL 35812

Final Report on:

Contract No. NAS8-36955, Delivery Order No. 58
Period of Performance: 17 October 1989 to 16 July 1990

Disclaimer Statement:

"The views, opinions, and/or findings contained in this report are those of the authors and should not be construed as an official NASA position, policy, or decision, unless so designated by other documentation".

Distribution Statement:

Distribution is unlimited.

(NASA-CR-184417) ENGINE DATA INTERPRETATION SYSTEM (EDIS) Final Report, 17 Oct. 1989 - 16 Jul. 1990 (Alabama Univ.) 161 p CSCL 21H

Unclas
G3/20 0003703

UAH Research Report No. ME-90-101
Date of Issue: July 1990

ENGINE DATA INTERPRETATION SYSTEM (EDIS)

Prepared by:

Thomas L. Cost and Martin O. Hofmann
College of Engineering
The University of Alabama in Huntsville
Huntsville, AL 35899

Prepared for:

George C. Marshall Space Flight Center
National Aeronautics and Space Administration
Marshall Space Flight Center, AL 35812

Final Report on:

Contract No. NAS8-36955, Delivery Order No. 58
Period of Performance: 17 October 1989 to 16 July 1990

Disclaimer Statement:

"The views, opinions, and/or findings contained in this report are those of the authors and should not be construed as an official NASA position, policy, or decision, unless so designated by other documentation".

Distribution Statement:

Distribution is unlimited.

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
UAH-TR-ME-90-101		

4. Title and Subtitle	5. Report Date
ENGINE DATA INTERPRETATION SYSTEM (EDIS)	July 1990
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Thomas L. Cost and Martin O. Hofmann	10. Work Unit No.

9. Performing Organization Name and Address	11. Contract or Grant No.
The University of Alabama in Huntsville College of Engineering Huntsville, AL 35899	NAS8-36955; DO #58
	13. Type of Report and Period Covered FINAL 17 Oct '89 to 16 July '90

12. Sponsoring Agency Name and Address	14. Sponsoring Agency Code
NASA/MSFC Propulsion Laboratory MSFC, AL	

15. Supplementary Notes

16. Abstract

A prototype of an expert system has been developed which applies qualitative or model-based reasoning to the task of post-test analysis and diagnosis of data resulting from a rocket engine firing. A combined component-based and process theory approach is adopted as the basis for system modeling. Such an approach provides a framework for explaining both normal and deviant system behavior in terms of individual component functionality. The diagnosis function is applied to digitized sensor time-histories generated during engine firings. The generic system is applicable to any liquid rocket engine but has been adapted specifically in this work to the Space Shuttle Main Engine (SSME). The system is applied to idealized data resulting from turbomachinery malfunction in the SSME.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Expert System. Artificial Intelligence. Engine diagnostics. SSME. Data interpretation.	Distribution unlimited.

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
UNCLASSIFIED	UNCLASSIFIED	110	

NASA FORM 1626 OCT 86

SUMMARY

A prototype of an expert system has been developed which applies qualitative or model-based reasoning to the task of post-test analysis and diagnosis of data resulting from a rocket engine firing. A combined component-based and process theory approach is adopted as the basis for system modeling. Such an approach provides a framework for explaining both normal and deviant system behavior in terms of individual component functionality. The diagnosis function is applied to digitized sensor time-histories generated during engine firings. The generic system is applicable to any liquid rocket engine but has been adapted specifically in this work to the Space Shuttle Main Engine (SSME). The system is applied to idealized data resulting from turbomachinery malfunction in the SSME.

CONTENTS

1. INTRODUCTION

The task of post-test analysis and diagnosis of data generated during rocket engine firings requires considerable labor by a team of experts. Data in the form of sensor histories displayed in graphical formats are perused to determine if the engine firing was "as expected" or anomalous. If an anomaly is suspected, attempts are made to identify the cause of the anomaly. When engine firings are conducted on a two-to-three day cycle, the team of experts can be occupied almost continuously in data review - these same experts are normally urgently needed to perform other tasks at the time of the data review. An exacerbation of the manpower problem is caused when experts retire or otherwise leave the team - their replacement is difficult.

To help alleviate the manpower problem associated with data review, an automated system is needed to provide assistance to the experts. Such a system would be implemented on a digital computer, would be capable of analyzing digitized test data - identifying normal and anomalous firing data, and would be capable of formulating hypotheses about the cause of the anomalous results. Furthermore, the system should be capable of justifying and explaining the stated hypotheses and recommending further actions to better identify the causes of the anomalies.

Current research in the development of diagnostic systems for rocket engine firing focuses on such approaches as expert systems [1,2], neural networks [3-6], and signal processing [7-9]. Traditional expert systems developed from the associational knowledge of human experts tend to have a very narrow scope both in terms of the extent of the domain and range of problem solving activities they can handle. Also, such systems do not provide sufficient flexibility for system modification - modification of the object of interest often calls for the development of a new expert system.

Model-based approaches [10-12] which integrate fundamental principles, causal and common sense knowledge are capable of overcoming the limitations of traditional expert systems. Several recent applications of qualitative or model-based diagnostic approaches appear applicable to the task at hand - the analysis and diagnosis of rocket engine data [13,14].

In what follows, a description of the model-based approach is discussed in a generic sense prior to application of the concept to the SSME system.

2.0 REVIEW OF DIAGNOSTIC APPROACHES

Before focusing on the particular application (SSME) and diagnostic system (EDIS) of primary interest here, a general review of the approaches other researchers have pursued seems appropriate. Relevant "Artificial Intelligence" (AI) literature includes previous work on knowledge-based analysis and diagnosis of the SSME and other space-related engineering systems. In what follows, approaches taken by researchers considering diagnostic systems similar to the SSME are summarized.

2.1 Generic Diagnostic Paradigm

Work on diagnostic systems by Davis [13] and Genesereth [15] and promising results in reasoning from first principle [16] have made model-based reasoning an attractive option for diagnostic systems. In particular, model-based reasoning allows diagnosis to be performed without explicit fault assumptions. A fault is simply characterized by a component not behaving as desired without reference to a specific aberration, see Davis [17]. Constraints are used to specify correct component behavior. A constraint is a qualitative or quantitative relationship between the parameters which describe the behavior of a component. A component fault can thus be defined as the violation of one or more constraints associated with the component. Model-based diagnosis using constraint propagation potentially covers all possible faults of a device, not only those explicitly enumerated by an expert. Diagnostic completeness is, however, limited by the accuracy and completeness of the model [17]. For example, parasitic causal pathways may exist between components, such as heat transport or crosstalk, which cannot be detected if the relevant kind of component interaction has not been modeled.

Diagnostic paradigms have been formulated based on the availability of a model which contains device structure, i.e., a decomposition of the device into interconnected components, and behavior constraints for all components of the device. For example, Davis [17] introduced "constraint suspension" and Genesereth's DART [15] program uses the "resolution residue" procedure. Most diagnostic procedures follow the Generate - Test - Discriminate paradigm. In the first step, fault hypotheses are generated. A hypothesis may explicitly enumerate a specific set of components which are assumed to be faulty, or a hypothesis may be implicitly defined by a set of components at least one of which must be faulty. The set of hypotheses must be complete but it will, in general, contain too many candidates, although most designs try to keep the set as small as possible. Hypothesis testing eliminates those candidate hypotheses which cannot account for all observed symptoms.

Theoretically, it is possible to combine hypothesis generation and testing, i.e. to generate viable hypotheses only, but in practice it often proves simpler to separate these two steps. If several hypotheses survive testing, then more data need to be observed to discriminate between them. Electronic troubleshooting systems must determine the test which promises to reveal the most new information. The FIS system [18], the IN-ATE approach [19], and the general diagnostic engine (GDE) method [20] use probabilistic methods to propose the next "best" test. Approaches based on the minimum entropy principle, such as GDE, appear to be best. Note, however, that for SSME post-test analysis no further tests are possible.

DeKleer and Williams [20] have presented GDE, a method for diagnosing single and multiple faults in systems which can be modeled by interconnected modules, each characterized by constraints between input and output parameters. Essentially the same method has also been proposed by Reiter [21] except that his derivation is based on formal logic. GDE predicts values for device parameters given some known values, e.g., measured or input values, by propagating the known values through the component interconnections and constraint expressions. Note that constraints must be non-directional, i.e. the system must be able to reason from inputs to outputs as well as from outputs to inputs. Davis [13], for example, supplies "simulation" and "inference" rules for forward and backward propagation, respectively.

GDE detects a "symptom" when at least two different values are predicted (or determined) for the same parameter based on different input or measured values. Value prediction depends on the assumption that each component which was traversed during constraint propagation enforces its constraints correctly. Existence of a symptom indicates that at least one constraint must be violated and thus one component involved in the symptom must be faulty. A component is involved in a symptom if it lies on a propagation path which leads to the symptom, i.e. its behavior influences the predicted value. A symptom gives rise to a set of fault hypotheses. GDE represents hypotheses implicitly. Sets of components which contain at least one fault, named "conflicts" or "conflict sets", are generated by combining all components which were involved in creating the symptom. Hypotheses are derived from these conflict sets by forming sets of components such that at least one member of each conflict set is represented in the hypothesis set. If a hypothesis exists which contains only one component then a single fault in this component can account for all symptoms. Otherwise multiple faults must be present.

The diagnostic paradigm exemplified by GDE is very powerful

but some caution is appropriate before recommending it for every diagnostic application. DeKleer and Williams [20] point out that complete prediction of component and system behavior is currently beyond the state-of-the-art. The SSME [22] is a good example of a complex dynamic system whose behavior is very difficult to model and to predict. A large numeric power-balance model (PBM) [23] is used for post-test data reduction and pre-test performance prediction. The PBM searches iteratively for a set of consistent engine parameter values. It is valid only for normal operation and some small deviations. Clearly, such a model cannot be used for constraint propagation, because it cannot propagate anomalous parameter values and because it cannot perform local propagation at each module.

2.2 Reduced Prediction Models

Given that the SSME components cannot be modeled by exact constraints, other, less accurate methods of modeling have to be explored. Qualitative modeling [16] eliminates the need for exact numeric constraint equations. Only qualitative parameter values, such as normal, low, high, and their trends, such as stable, increasing, decreasing, are considered. Several systems have been developed which can perform system simulation using qualitative models only, see [16]. In [24], for example, DeKleer and Brown define qualitative models for components as sets of qualitative state - confluence pairs. Qualitative states loosely correspond to operating regions of devices governed by different laws. Confluences are equations constraining qualitative values of parameters, based on a special qualitative calculus. Confluences and qualitative states are usually derived from conventional mathematical models.

Forbus [25] presents another approach to qualitative modeling which is process-centered instead of component-centered. A process relates the parameters of several interacting objects (components). For example, a heat flow process is instantiated when a heat source, a heat sink, and a heat path are present and properly aligned.

Qualitative models have the disadvantage that counteracting influences lead to multiple possible conclusions about the behavior of a parameter value. For example, if two input parameters are added to produce the output and the signs ("high" or "low") of the inputs do not agree, the sign of the output cannot be predicted uniquely. The same holds true for opposite trends at the inputs. Predictive ability is limited because the relative strengths of conflicting influences are not represented.

Subsequently, researchers have modified the concept of

qualitative modeling by replacing qualitative confluences by simplified analytic equations which allow exact comparison of conflicting influences and the use of known component parameters, such as efficiency coefficients. Govinderaj [26] describes a qualitative approximation methodology using "moderate fidelity simulators". System components are modeled using simplified dynamical equations abstracted from continuity and compatibility conditions. Biswas [10] describes another modeling methodology using analytic equations which approximate actual device behavior.

Even less information is required for causal modeling. Causal models, in their simplest form, only describe the causal relationships between aberrations of component behavior. Component behavior is abstracted into function and the functional model merely describes which functions, and therefore which components, depend on each other. All that can be said about a pump, for example, is that its function is to create a pressure increase, whether it performs this function or not, and which subsequent function depends on the correct functioning of the pump. An extended causal model will enumerate types of anomalies of functions and how anomalies in one component cause anomalies in the functions dependent on it. In a staged pump system, for example, reduced pump performance in the first stage will increase pump workload in the second stage.

Still more detail can be incorporated into a causal model if deviations of parameter values are considered instead of deviations in overall function. For example, anomalous pressure at the input of a pipe will result in anomalous pressure at its output. Low input pressure to a pump leads to low output pressure unless a controller increases the power driving the pump. This detailed causal model approaches the capabilities of a qualitative model, except that it describes deviations from a norm instead of absolute behavior. Govinderaj's system [26] also reasons about deviations from steady-state but uses quantitative equations. He advocates his approach for applications involving large complex dynamic systems such as a marine steam power plant.

A most interesting aspect of functional models is the possibility to switch between levels of abstraction and relate the functioning of a component to the functioning of the enclosing module. The intrinsic function of a pump is to expel fluid at a pressure higher than at the intake, while in the context of the SSME, the function of the pump may be to push fuel through the cooling circuits at a high enough rate. Sembugamoorthy and Chandrasekaran [27] and Bylander [28] have presented an approach to this problem but more needs to be done.

2.3 Alternative Diagnostic Methods

As suggested by the classification scheme for diagnostic systems delineated by Milne [12] we will discuss compiled knowledge systems in the following, having completed the presentation of structural, behavioral, and functional models.

Abstracting device behavior and function beyond causal models leads to "compiled" diagnostic systems which explicitly associate symptoms with fault hypotheses. Heuristic, pattern matching, or associational systems belong to this category. Most commercial expert systems are based on compiled heuristics and specialized software tools have been developed to help build them. Frequently, heuristics are stored as production rules. The validity of a compiled system depends on completeness of the rule base and exhaustive enumeration of possible faults. Rules can be created by experts or extracted from case data.

The advantages of heuristic based systems are that they can deal with common faults rapidly and economically, that they do not need good models of the device, and that the user group is more likely to accept a knowledge-based system if they were involved in its creation. The disadvantage of expert systems based on application-specific heuristics are that they only cover explicitly enumerated faults, that they are difficult to maintain and extend, and that they apply only to a specific application.

Continuing research on compiled knowledge systems has generated approaches to generalize and reuse heuristics from one application to another, see, for example, Malin and Lance [29]. Generalization of heuristics which are tied to particular components requires reversing the symptom-fault heuristic to a fault-symptom prediction format. Component models which are to produce heuristic rules thus need to facilitate enumerating the possible faults of a component and to predict the effects of those faults on component behavior. These models differ from the models discussed above in that they contain knowledge about specific faults and effects of faults. Of course, they are also used differently, i.e. to create heuristic rules which embody symptom-fault associations.

2.4 Use of Fault Models

Fault models, i.e. descriptions of how the behavior of a component changes given a fault has occurred, have the potential to assist in selecting fault candidates, testing fault hypotheses, and refining fault hypotheses. Substantial differences in the use of fault models warrants a more detailed analysis of the utility of fault models. Some model-based

systems, such as GDE, operate totally without resorting to the use of fault models. They operate under the assumption that hypotheses can be pruned and refined by collecting additional data until a unique fault (or set of faults) has been determined.

Fault models may be used to determine if a candidate component, in fact, has a failure mode which can account for the observed symptoms. This method can be applied in model-based systems when several competing hypotheses remain but no further data can be collected to discriminate between them. At this point some assumptions must be made in order to proceed with the diagnosis. Using fault models to eliminate hypotheses implies the assumption that the enumerated fault modes are more likely to occur than other, as yet unconceived, faults.

Fault models can also refine a unique hypothesis by postulating a particular fault in a component. Generic diagnosis, such as the GDE methodology, pinpoints only a component, but does not identify how it has failed. If the actual fault is of interest, or if the fault is to be localized more precisely within the component but no detailed component model is available, then fault models can be matched against the observed symptom.

2.5 Mixed Paradigm Systems

Model-based and heuristic-based diagnostic systems each have unique advantages and disadvantages as discussed above. To incorporate both paradigms into one system could potentially combine the strengths of each approach. Establishing smooth cooperation between these divergent methods poses some problems, however. Model-based systems execute in a sequential, algorithmic manner, where hypotheses are first generated, then tested, and finally discriminated. Heuristic rule-based systems for the most part operate in a goal-driven associational fashion. Hypotheses are created one at a time; each is evaluated separately using observations or intermediate inferences left over from processing a previous hypothesis. Hypotheses may be discarded any time the conditions of a rule are satisfied by some pattern in the data.

Typically, systems which incorporate model and heuristic-based reasoning alternatively execute two separate reasoning mechanisms for each paradigm. For example, Fink [30] describes the IDM (integrated diagnostic model) system which first executes a heuristic module and then switches to a model-based module when the heuristics fail to provide a diagnosis. A conversion mechanisms is provided which allows sharing of information between the two modules. Another

approach, specified by Pazzani and Brindle [31], calls on heuristic rules to hypothesize faults and device models to confirm or deny those hypotheses. Pflueger [32] mixes experiential diagnosis based on associational rules and model-based reasoning using a "logic function model" and constraint propagation and suspension. Rules are used to accelerate recognition of frequent faults and in cases where components cannot be adequately modeled.

3. DESCRIPTION OF SPACE SHUTTLE MAIN ENGINE SYSTEM

The Space Shuttle Main Engine (SSME) is a reusable, high performance, liquid-propellant rocket engine with variable thrust. Figure 1 contains a schematic diagram of the main components of the engine. The engine burns liquid oxygen and liquid hydrogen at a mixture ratio of 6:1 to produce a sea level thrust of 375,000 pounds. The chamber pressure is approximately 3000 psia and the SSME is throttleable over a range of 65 to 109 percent of rated power level. The engine is regarded as a high-performance engine due to the high chamber pressure and the use of a staged combustion power cycle.

In the SSME staged combustion power cycle, the propellants are partially burned at low mixture ratio, very high pressure, and relatively low temperature in the preburners to produce hydrogen-rich gas to power the high-pressure turbopumps. This hydrogen-rich steam is then routed to the main injector where it is injected along with additional oxidizer and fuel into the main combustion chamber. Hydrogen fuel is used to cool all combustion devices directly exposed to high-temperature products of combustion. An electronic controller automatically performs checkout, startup, mainstage, and shutdown operations.

3.1 Major Components

Key components to the SSME system are four turbopumps, two low pressure and two high pressure:

1) low-pressure fuel turbopump (LDFTP)
2) low-pressure oxidizer turbopump (LPOTP)
3) high-pressure fuel turbopump (HPFTP)
4) high-pressure oxidizer turbopump (HPOTP)

These pumps are identified in Fig. 1.

The LPFTP and LPOTP are axial-flow pumps that operate at relatively low speeds and provide the pressure increase required at the inlets of the respective high-pressure turbopumps.

The HPFTP is a three-stage, centrifugal-flow pump driven directly by a two-stage hot-gas turbine. The HPOTP consists of two centrifugal-flow pumps on a common shaft and driven directly by a two-stage hot-gas turbine. The main pump supplies oxidizer to the main combustion chamber, the LPOTP turbine, and the preburner oxidizer pump. The preburner oxidizer pump raises the pressure of the oxidizer and supplies it to the fuel and oxidizer preburners.

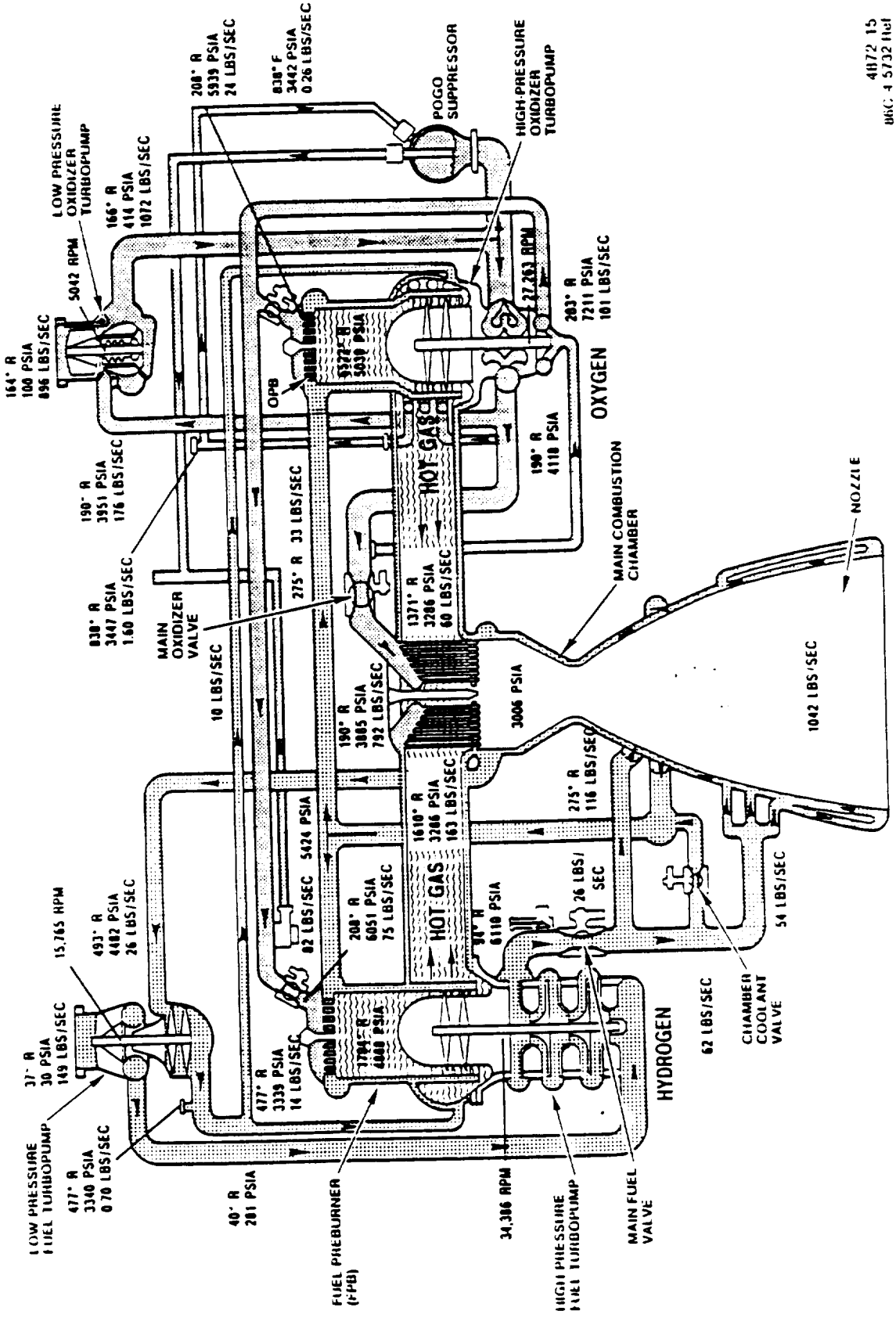


Figure 1. Schematic of SSME.

ORIGINAL PAGE IS
OF POOR QUALITY

The hot-gas manifold (HGM) is the structural backbone of the SSME engine system in that it supports two preburners, two high-pressure pumps, the main injector and the main combustion chamber. It interconnects the fuel and oxidizer preburners (FPB and OPB) to the main chamber injector. The FPB and OPB generate fuel-rich gases that power the HPFTP and HPOTP.

The main combustion chamber (MCC) is attached to the HGM and consists of an internal coolant liner and an external structural jacket. The nozzle is bolted to the MCC.

In addition to the items mentioned above, the SSME key components are connected by various interconnects: main propellant articulating ducts, fluid interface lines, and component interconnects. These interconnects contain important valves such as the main oxidizer valve (MOV), main fuel valve (MFV), fuel preburner oxidizer valve (FPOV), oxidizer preburner oxidizer valve (OPOV), and the chamber coolant valve (CCV).

For simplicity, the SSME system considered in this work has been simplified by omitting the pogo suppression system, the propellant tank pressurization system and certain minor propellant ductwork - none of these omissions change the basic operation of the system. The simplified system is illustrated in Fig. 2.

For the purposes of system modeling, these components need more definition. This definition is provided in Section 4.3.

3.2 Interconnectivity

The interconnectivity of the SSME system key components is illustrated schematically in Fig. 2. As can be seen the turbines and pumps are directly (mechanically) connected, the preburners are directly connected to the respective turbines and all other components are connected by propellant ducts. Precise statements of interconnectivity are described in Section 4.4.

3.3 Test Data

Test data from SSME engine firings are recorded as analog signals on magnetic tape and later digitized and stored in files on hard disks. The data is in the form of time-histories of individual sensor output. The EDIS system accesses these digitized data files and performs diagnostic functions by comparing data values with expected or calculated values.

Simple temperature, pressure, and shaft speed

ORIGINAL PAGE IS
OF POOR QUALITY

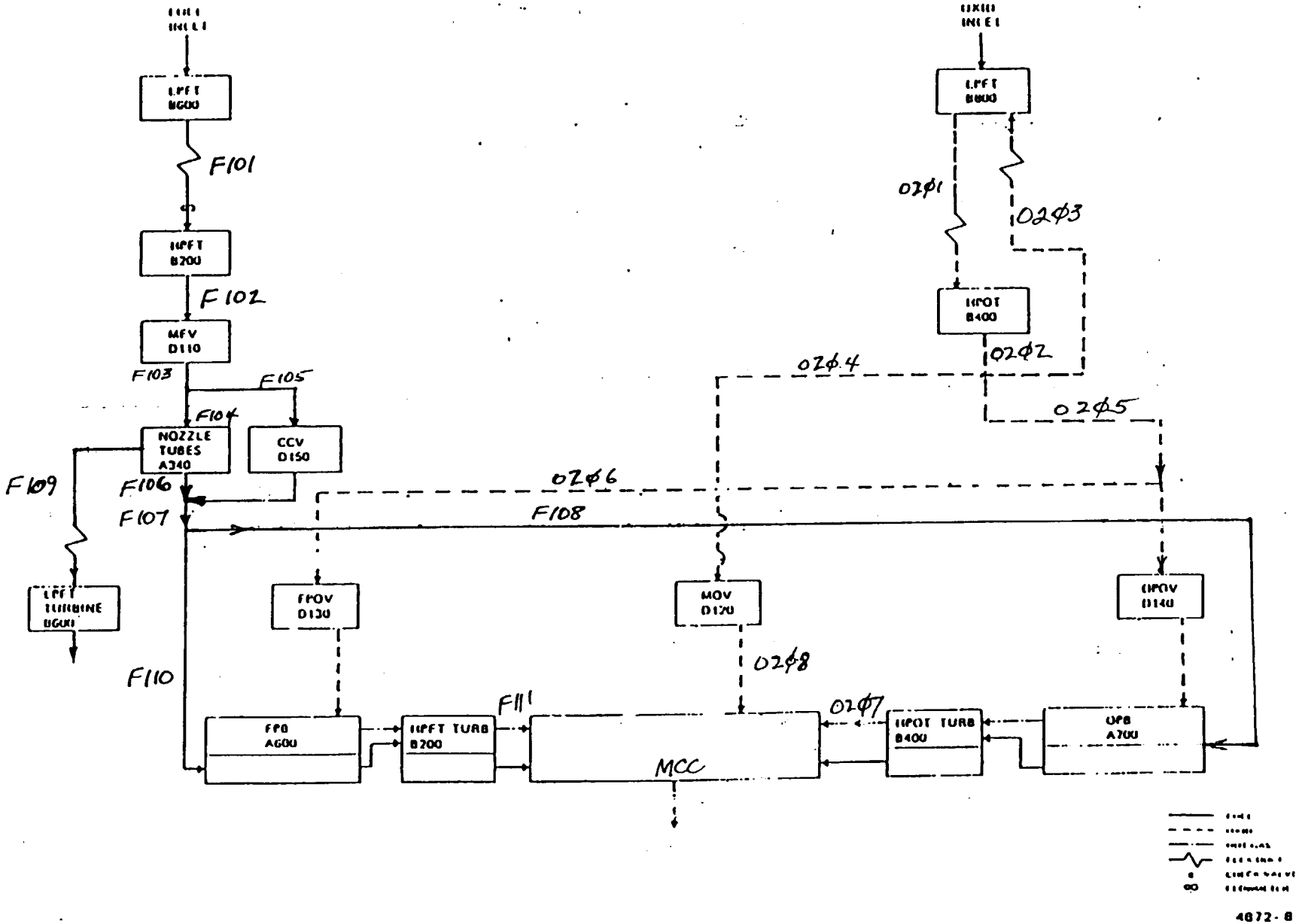


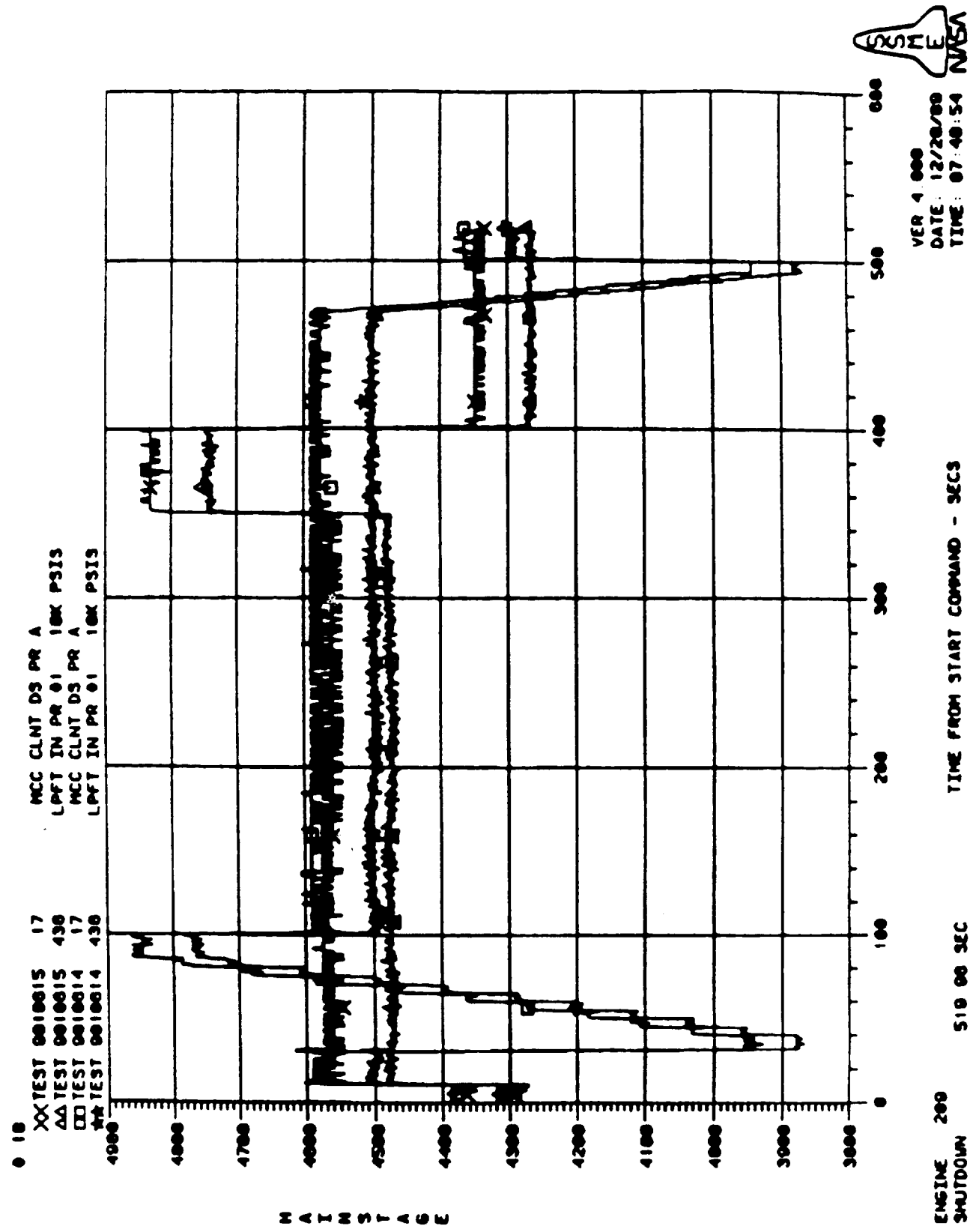
Figure 2. SSME Propellant Block Diagram
(for reference only)

Summary

Fuel Lines	Oxidizer Liner
11	8

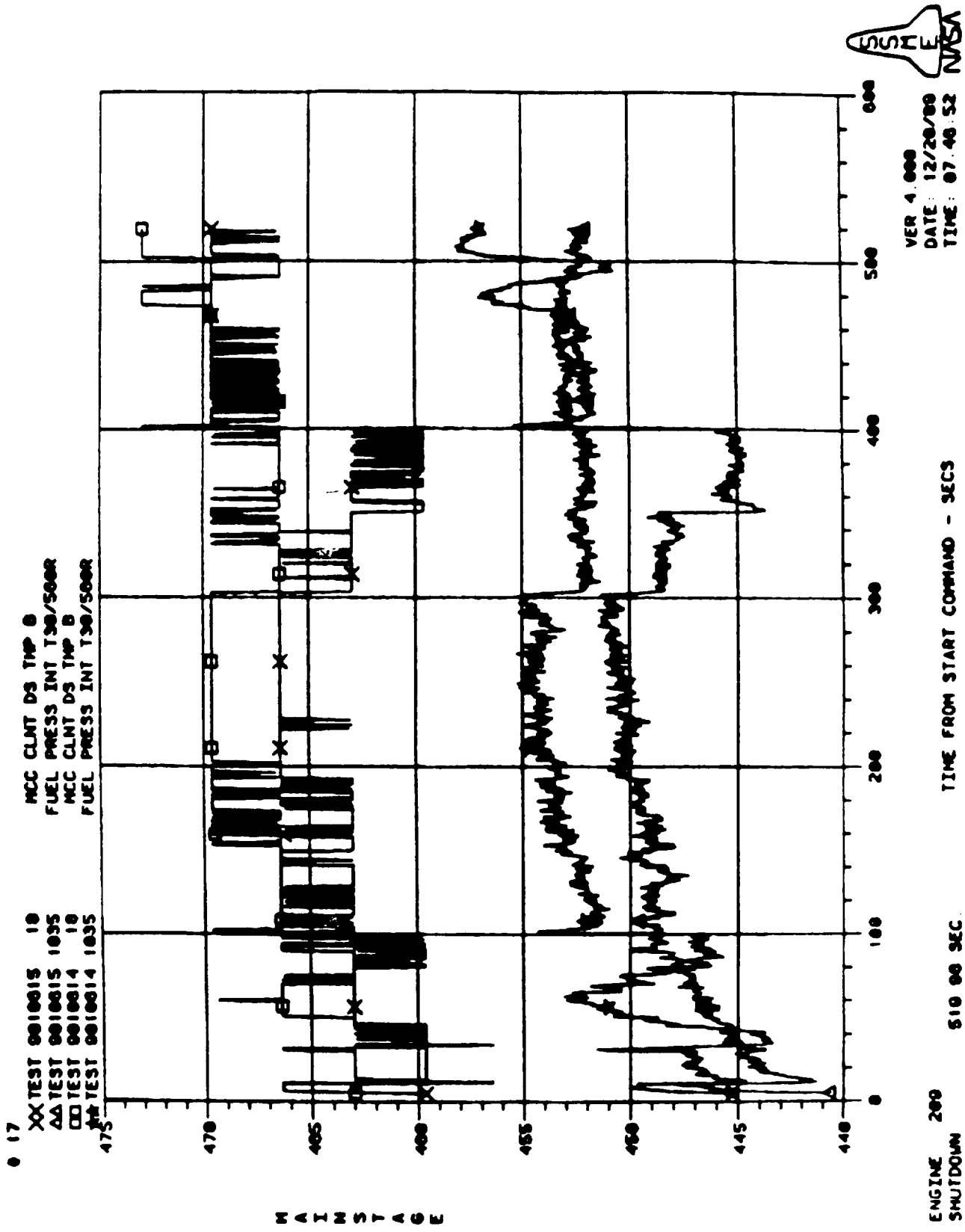
time-histories from a typical SSME static firing are illustrated in Figures 3, 4, and 5. As can be seen, the data contains engine start, mainstage, and shutdown phases. Current EDIS operation is restricted to consideration of only the mainstage of operation.

Figure 3. Temperature History During SSME Firing.



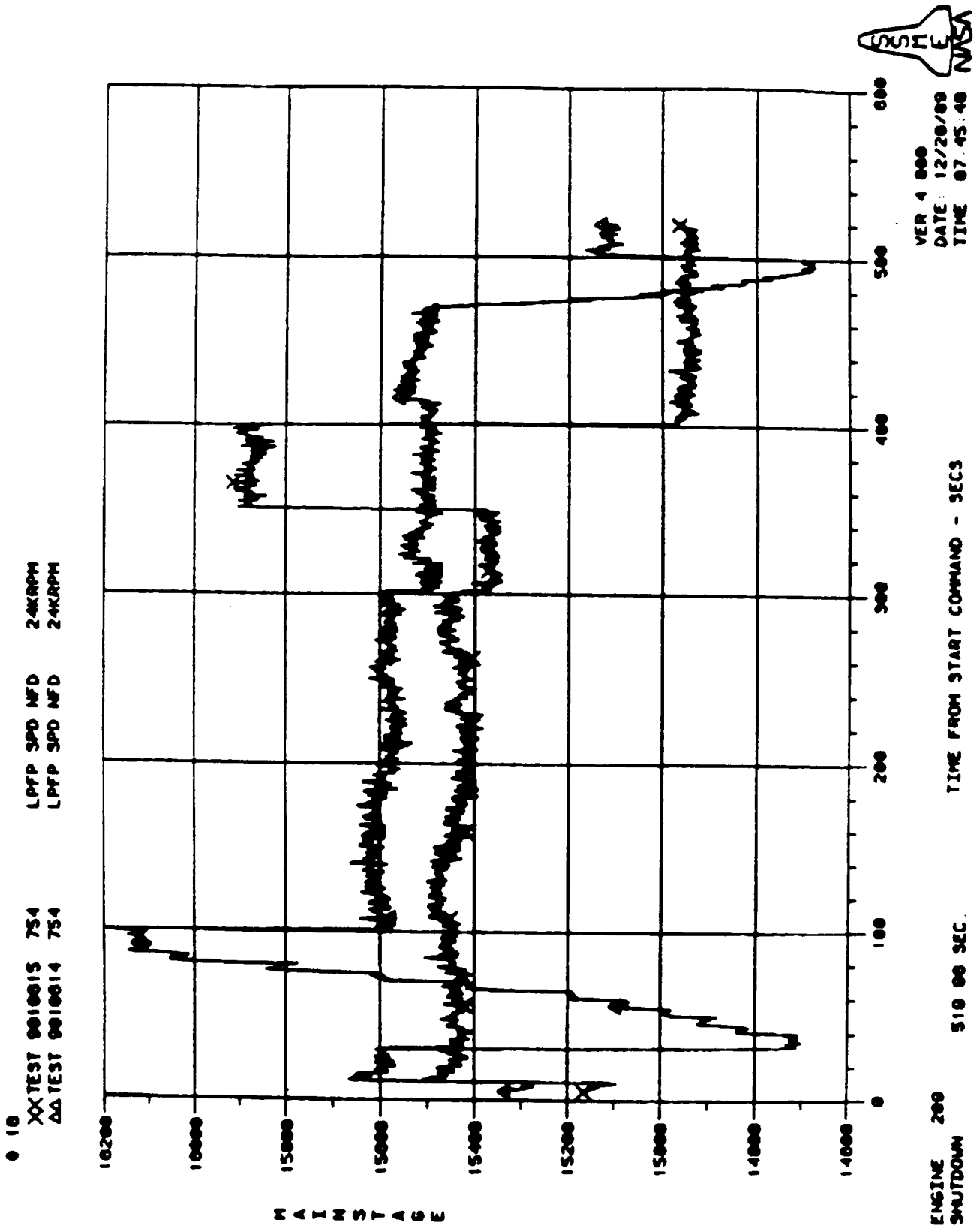
ORIGINAL PAGE IS
OF POOR QUALITY

Figure 4. Pressure History During SSME Firing.



ORIGINAL PAGE IS
OF POOR QUALITY

Figure 5. Shaft Speed During SSME Firing.



ORIGINAL PAGE IS
OF POOR QUALITY

4. ENGINE DATA INTERPRETATION SYSTEM (EDIS) DESCRIPTION

4.1 Diagnostic Paradigm

Goals - Selection of a diagnostic paradigm to be integrated into the SSME data review process was constrained by a number of goals specified at the outset of the project. The finished system was to contain generic propulsion and engine operation knowledge and to be configurable for various engines and engine variations. The SSME itself is undergoing continual modifications which must be accounted for by a diagnostic system. The diagnostic operation should be easy to modify and upgrade in order to provide a stable platform for future enhancements. The system should be able to explain its reasoning steps in terms and formats familiar to the user. The reasoning process should be controlled by an explicit strategy module which affords the user the opportunity to change and direct diagnostic reasoning. The system should be able to use available numeric engine models and records of past engine performance. The diagnostic paradigm developed in this work addresses these issues. It will be described below.

Specific Considerations - Diagnosis of the SSME differs in some important aspects from diagnosis of devices as commonly reported in the literature. The SSME is a complex system, and therefore difficult to model, not because it has a large number of components but because the thermodynamic processes are non-linear and coupled, and because some of its parameters are regulated by an engine controller. The controller will not allow deviations of controlled parameters within the limits of its capability. Deviations will show up at the actuated variables instead.

Testing an SSME is very complicated, labor-intensive, and expensive. It is not possible to repeat a test to get more or different measurements due to the limited life of individual components and the unique conditions surrounding each test. The question of selecting additional points to probe is mute. To offset the lack of additional measurements an unusually large number of parameters are measured during each test. In many cases redundant instrumentation measures the same parameter. Access to an almost complete set of test data is beneficial. Nevertheless, the amount of data recorded during one test makes it difficult for the reviewers to select relevant information from the bulk of data.

Due to the lack of a simple and accurate engine model, the data review process is largely based on comparing test data to records of previous tests, to average and normal variation data, and to absolute limit data. These comparison data are stored in

databases and can be plotted for visual comparison during the review process. Some of the historic data records were measured on engines which turned out to be defective. These records can be compared to new records when the same fault is suspected to be present in the current test. Available numeric engine models are executed in order to quantitatively characterize engine and turbomachinery performance and sometimes to predict effects of faults. Fault prediction is limited to small fluid and gas leaks and pipe obstructions.

Diagnosis is performed in the context of the data review process only, i.e. off-line. Real-time operation of the diagnostic system is not envisioned at this time, especially since interaction with review personnel is required. Ways of adapting the diagnostic paradigm to on-line monitoring and diagnosis may be investigated later.

Design - The SSME review process is composed of several tasks. First, test data are inspected to detect data anomalies. Anomalies are then characterized according to whether they are value, i.e. static, or dynamic deviations, whether they occur during start-up, main-stage, or shutdown, and whether they are consistent or erratic. Anomaly explanation is based on the experience that anomalies can be caused by sensor problems, data manipulation and presentation artifacts, and by actual engine-related causes. The SSME will produce slightly different data at every test because of random variations, because of wear in the turbo-machinery, and because of replaced turbo-machinery. Actual engine problems can be related to turbo-machinery alone or to faults somewhere else in the engine. Finally, engine behavior may deviate from the norm because of changes in throttle control demanded by special test objectives.

A knowledge-based system to support the review process must take all these real and pseudo-faults into account when interpreting data anomalies. Our design calls for the following steps. Anomaly detection, verification, and fault diagnosis. Anomaly verification eliminates deviations due to test objectives, data manipulation, random variations, and sensor problems from consideration. Fault diagnosis finds turbo-machinery and general engine faults. At this point only fault diagnosis has been developed in detail. The diagnostic method is described in the next section.

The above described method of first classifying anomalies into one of several categories is an example of diagnosis by successive refinement or focusing. Chandrasekaran [33] has identified hierarchical, successive refinement as one of a few generic reasoning methods. Hierarchical diagnosis (or problem solving in general) is commonly used by human experts [34]

because it reduces the complexity of diagnostic search. The diagnostic method loses some generality, however, when anomaly explanations are separated into classes. Constraint propagation techniques, for example, can find multiple faults only under the condition that a single complete model exists which fully describes the system to be diagnosed. Unfortunately, it is hard to imagine a model which can combine physical descriptions, control variations, and data manipulation procedures. Sensor behavior could be incorporated fairly easily, though.

Method - The diagnostic procedure is compartmentalized into hypothesis generation, testing, and discrimination. The architecture of the diagnostic system, see Section 4.2, provides means to explicitly represent anomalies, hypotheses, and decisions about hypotheses, as well as means to dynamically schedule knowledge sources. These architectural features make it possible to combine and coordinate various diagnostic paradigms. For example, hypotheses can be created by a constraint propagation mechanism as in GDE [a], by heuristic rules contributed by a human expert, or by rules induced from exhaustive fault simulation. In every situation the most appropriate paradigm can be chosen in order to maximize system performance. In addition, hypotheses can be formulated and examined with the help of numeric engine models and records of previous test data.

Constraint-based diagnosis based on a causal model of the SSME is the primary method which ensures maximum fault coverage. In Section 4.3 we will present the constraint propagation mechanism and the qualitative model in detail. Heuristic rules acquired from human experts are included to serve two purposes. Rules are able to identify common faults quickly and they can be applied to discriminate between hypotheses when not enough data are available to disambiguate the diagnosis. Moreover, we plan to incorporate a robust rule acquisition mechanism which will allow experts and prospective users to add heuristics to the system. This will, we hope, increase acceptance of the system for routine use.

Hypothesis Generation - Constraint propagation in the qualitative model and heuristic rules generate hypotheses. Hypotheses created by constraint propagation are consistent with the observed symptoms but not necessarily with the expected fault modes of components. Hypotheses constructed by heuristic rules may not be consistent with the symptoms or the fault modes. Their validity depends totally on the quality, i.e. correctness, consistency, and completeness, of the expertise incorporated in the rules. Quality has to be assured during the knowledge acquisition process.

Hypothesis Testing - The validity of hypotheses created by constraint propagation depends on the accuracy of the qualitative model. Some hypotheses produced by a model of little detail can be eliminated when a more detailed model is consulted. For example, hypotheses generated from a model based on qualitative relations only, may be tested with the help of a simplified quantitative model which characterizes anomalies and behavior more accurately. Chances of eliminating valid hypotheses are negligible unless the model is overly simplified. Tests may also be based on physical plausibility, e.g. conservation laws. For example, a component cannot exhibit a fault mode where energy is created.

Hypothesis Discrimination - When several hypotheses remain after testing, hypotheses are ranked according to plausibility. Fault plausibility is increased by agreement with numeric fault simulations, by correlation with predetermined fault models, by agreement with previous anomaly - fault observations, and by observed frequency of occurrence of the fault. The final result of diagnosis is a ranked list of plausible fault hypotheses which could not be ruled out. In general, no single unique fault can be determined.

4.2 Architecture

EDIS is built upon a modular blackboard architecture. EDIS system modules are defined and implemented independently from each other, lending flexibility to system development, enhancement, and maintenance. The EDIS system is modularized according to functional criteria which do not necessarily reflect physical modularization. Functional modularization facilitates intelligent scheduling and allows the user to actively participate in the problem solving process via a mixed-initiative dialogue. Major functional units include data retrieval, sensor validation, diagnosis, and user interfacing. Functional units may be decomposed into smaller tasks. The diagnostic process, for example, is subdivided into anomaly detection and classification, hypotheses generation, hypotheses testing, and hypotheses discrimination. All of these modules operate on the whole SSME model because behavior of SSME components cannot, in general, be evaluated in isolation.

All EDIS modules share a common explicit structural and functional model of the SSME. More specialized models, such as turbo machinery models and combustion process models, for example, may reside within individual modules. Reasoning based on these special purpose models is separate from the basic diagnostic process. A similar separation has been observed to exist in the current data review process where turbo machinery

and other specialists are generally only consulted to verify hypotheses created from analysis of engine performance.

The blackboard serves at the same time as central inter-module communication medium and as repository for system state information. The blackboard is the common communication medium through which all modules exchange information. Modules encapsulate reasonably self-contained functions so that the need for inter-module communication is minimized. Anomalies, hypotheses, and other important items are stored explicitly on the blackboard. There they can be read by other modules and the blackboard serves as a communication medium. At the same time, however, the information stored on the blackboard represents the state of the analysis process since findings, hypotheses, and also tasks (previously executed as well as scheduled ones) can be found there. Normally, information is never deleted from the blackboard. Instead, items are marked as obsolete when necessary. Obsolescence decision time and agent are recorded with the item. Decisions about data validity are thus made explicit and reversible.

A complete, explicit account of system state makes in-depth explanation of system actions and reasoning possible. Explanations can be prepared according to current system goals and against the background of previous decisions and events. Explanation becomes independent from specific reasoning implementations, such as rules, and even reasoning mechanisms. Conclusions, decisions, and supporting information can be examined instead of rules.

Module functions can be classified into control, diagnostic reasoning, data interface, and user interface functions. A strategy module controls the scheduling of all other modules. It is scheduled automatically when the EDIS system is first initialized. The strategy module creates tasks on the blackboard which identify the modules (also called knowledge sources in the context of blackboard management) to be executed. The strategy module can schedule itself repeatedly to monitor the progress of data analysis and to possibly reschedule tasks. User interface tasks present data to users and ask for input. Both textual and graphical displays are available on the PC platform. For example, the user completes an input form to supply test and data file names and the type of comparison data used for anomaly detection. In later stages of reasoning anomalies, hypotheses, and inferences can be presented and verified or rejected by the user. A graphical representation of SSME structure helps visualize hypothesized causal relations between symptoms and faults. Specialized interface modules can be provided to access any of the various data bases which contain perfromance, configuration, and fault data.

The blackboard data structures mirror the object-oriented data structures of the reasoning modules. The blackboard contains data in the form of classes, class members, attributes, and attribute values. Classes define structure and attributes of their members. Attributes have names and values. Values are stored as character strings in order to be compatible with the knowledge engineering tool used (KES). Blackboard data structures are isomorph to KES class definitions. The blackboard assumes, however, that data are correctly formatted, i.e. no syntax checking is performed.

Blackboard data reflect the functional diversity of the system modules and can be classified into control, model, and reasoning data and parameter or input values. Control information constitutes the link between modules (especially the strategy module) and the system framework. Control information is interpreted by the scheduler and dispatcher which generate the actual flow of program execution. Members of the classes TASK and KNOWLEDGE SOURCE represent control information on the blackboard. A task is characterized by the attributes name, priority, and knowledge source. Task priority guides the scheduling mechanism in selecting the next task to execute. The specified knowledge source indicates

4.3 Domain Modeling

Following Biswas [10], the structural schematic of the SSME system is described in terms of primitive components, complex components, component categories, a set of interconnections, and fundamental processes. Table 1 contains a list of primitive components for the SSME system and Table 2 the primitive component categories. All turbopumps are considered to be complex components consisting of turbine and pump primitive components. Note that a distinction is made betwen gas-turbine and hydraulic-turbine turbopumps - different thermodynamic relations are used to describe the behavior of gases and liquids in turbine processes.

SSME structure is modeled as a collection of interconnected instances of components, each characterized by a generic thermodynamic process, see Section 4.4. SSME behavior is modeled in terms of deviations of engine parameter values from normal values. Sets of normal values have been collected at NASA MSFC for each operating region and are available in a data base. Deviations of parameter values can be propagated through the component network using component behavior models. Component behavior is modeled using constraints at two levels of specificity. A purely qualitative model is valid for any component of a given type, e.g. pump, pipe, etc. It relates

TABLE 1. LIST OF PRIMITIVE COMPONENTS

NAME		DESCRIPTION
LPFTT	-	Low Pressure Fuel Turbopump Turbine
LPFPP	-	Low Pressure Fuel Turbopump Pump
LPOTT	-	Low Pressure Oxidizer Turbopump Turbine
LPOTP	-	Low Pressure Oxidizer Turbopump Pump
HPFTT	-	High Pressure Fuel Turbopump Turbine
HPFTP	-	High Pressure Fuel Turbopump Pump
HPOTT	-	High Pressure Oxidizer Turbopump Turbine
HPOTP	-	High Pressure Oxidizer Turbopump Pump
FPB	-	Fuel Preburner
OPB	-	Oxidizer Preburner
MCC	-	Main Cumbustion Chamber
MFV	-	Main Fuel Valve
MOV	-	Main Oxidizer Valve
FPOV	-	Fuel Preburner Oxidizer Valve
OPOV	-	Oxidizer Preburner Oxidizer Valve
CCV	-	Chamber Coolant Valve
MCON	-	Controller
NOZ	-	Nozzle
F101	-	Fuel Duct 101
F102	-	Fuel Duct 102
F103	-	Fuel Duct 103
.		.
.		.
.		.
F111	-	Fuel Duct 111
O201	-	Oxidizer Duct 201
O202	-	Oxidizer Duct 202
.		.
.		.
.		.
O208	-	Oxidizer Duct 203
HY101	-	Hydrogen Fluid 101
HY102	-	Hydrogen Fluid 102
.		.
.		.
.		.
HY111	-	Hydrogen Fluid 111
OX201	-	Oxygen Fluid 201
.		.
.		.
.		.
OX208	-	Oxygen Fluid 208

TABLE 2. PRIMITIVE COMPONENT CATEGORIES

a) Gas Turbines
b) Hydraulic Turbines
c) Pumps
d) Ducts
e) Fluids
f) Valves
g) Preburners
h) Combustion Chambers
i) Controllers
j) Nozzles

qualitative deviations of input parameters to qualitative deviations of output parameters assuming the component is functioning correctly. Simplified quantitative models can be made available for the components of a particular system. Design parameters and empirically determined coefficients have to be incorporated into the quantitative equations. The quantitative model can determine and process relative strengths of influences.

In some cases local propagation results are indeterminate using either model. Indeterminacy is inevitable when parameter values depend on boundary conditions which can only be derived from an analysis of the complete system. Thermodynamic systems rarely exhibit unidirectional causality at the parameter level, i.e. parameter values almost always depend on the behavior of neighboring components and on boundary conditions. Also, component behavior is described by at least two or more interacting parameters, e.g. fluid or gas pressure, velocity, and temperature. When a constraint cannot be verified or used due to lack of data, the assumption is made that no or the smallest possible deviation from normal behavior has occurred. Assumptions are recorded and verified or rejected when new data become available, for example, during analysis of another component.

Fundamental constraints which describe correct component behavior are derived from energy conservation laws. When a constraint does not mention measurable parameters explicitly, normative constraints are added which hold under the assumption that the quantities on both sides of the fundamental constraint are constant. Normative constraints do not determine correct behavior but relate measurable parameters to fundamental constraints. They correspond to a more detailed model of the component in terms of thermodynamic processes. They organize the prediction/verification process so that behavior constraints can be verified incrementally and that necessary assumptions become evident.

Qualitative Behavior Model - Qualitative models consist of qualitative fundamental constraints, normative constraints, and auxiliary qualitative relations between quantities in different constraints. Constraints and relations determine existence and direction of the deviation in a quantity based on a deviation in a related quantity. Conceivably, deviations could additionally be characterized by qualitative statements of relative size but the current design does not use size. Constraints and relations are expressed in the same syntax. The general form of a qualitative statement is "Quantity-1 Relational-Operator Quantity-2". The two relational operators are "is proportional to" (p) and "is inversely proportional to" (ip). A quantity is

either a state parameter of the fluid or gas, such as pressure, a derived parameter, such as pressure difference, or an explicit measure of energy.

The semantics of fundamental and normative constraints and of auxiliary relations differ. A fundamental constraint captures an energy balance which must hold when the component is operating correctly. Faults are assumed to introduce additional losses, in general. Normative constraints must hold as long as the quantity they depend on remains constant. Auxiliary relations describe how a change in the presumably constant quantity (called a "pseudo-constant") are reflected in the quantities of the normative constraint. An auxiliary relation thus couples a normative relation to a fundamental relation via its pseudo-constant quantity. Normative constraints may be coupled to a fundamental constraint through a chain of other normative constraints in order to deal with more complex cases.

For example, the behavior of a pipe is characterized by the single fundamental constraint CP1: "Pressure-Difference p Velocity", meaning that the difference in fluid pressure measured at both ends of the pipe is proportional to the velocity of the fluid. This constraint was derived from the fluid energy balance neglecting possible differences in height and diameter of the pipe ends. The pipe has one normative constraint CP2: "In-Pressure p Out-Pressure" which holds (at least) as long as the pseudo-constant "Pressure-Difference" remains constant. One can observe that the normative constraint captures a superficial rule-of-thumb analysis of pipe behavior. Auxiliary relations are applied when the pseudo-constant has (or is suspected to have) changed and its changes have to be related to changes in the parameters of the normative constraint. In the example the auxiliary relations are "In-Pressure p Pressure-Difference" and "Out-Pressure ip Pressure-Difference", signifying that the pressure difference decreases with rising outflow pressure and decreasing inflow pressure.

The constraints associated with a pump are more complicated because energy is added to the system. In the case of the SSME energy is provided to each pump by its associated turbine. There are two fundamental constraints, one describing the transfer of mechanical energy from the outside (the pump shaft) to the fluid and one describing the transformation of fluid energy into a pressure difference. The fundamental constraints are CU1: "Mechanical-Power p E-V-Fluid" (E-V-Fluid refers to the fluid energy-velocity product) and CU2: "Fluid-Energy p Pressure-Difference". Constraint CU2 shares normative constraint and auxiliary equations with constraint CP1 described above in the context of the pipe model. Constraint CU1 has two

normative constraints associated with it, CU3: "Fluid-Energy ip Velocity" related to pseudo-constant "E-V-Fluid" and CU4: "Torque ip Shaft-Speed" related to pseudo-constant "Mechanical-Power". Constraints for other components are defined in a similar manner.

Simplified Quantitative Behavior Model - Simplified quantitative models have the same general structure as qualitative models. Fundamental equations express energy balances and normative equations define how a quantity (the associated pseudo-constant) in a fundamental equation can be determined from component parameters. Normative equations thus perform the function of both qualitative normative constraints and qualitative auxiliary relations. Therefore, auxiliary relations are not needed in the quantitative model. Constraints are expressed as analytic equations between parameters.

Simplified quantitative equations, i.e. constraints, are derived from exact thermodynamic equations neglecting as many terms as possible and performing linearization since the models describe deviations from the norm only. Equations are conditioned on a particular target system using application specific coefficients. Numeric coefficients can be determined from design specifications and from analysis of previous system performance. Some coefficients describe invariant properties of components, such as the friction coefficient of a pipe and should always remain constant. Other coefficients are variable, such as the efficiency of a turbopump which may change from one test to another. Limits on variation are imposed on non-constant coefficients instead of testing them against a single given value.

Reasoning With Models - The propagation process through the SSME model raises different issues as compared to a situation where few data are known to begin with. Propagation does not have to proceed across known values. Therefore the component network disintegrates into small subnets isolated by locations with known parameter values which can be analyzed individually. In fact, the decisions not to look beyond known values combined with not including conflicts based on two propagated values are equivalent to considering only minimal conflicts in GDE. There are some cases, however, where this strategy misses the real cause of the observed symptom, for example, when a component fails due to a fault at its input but masks the original fault. Only the secondary fault will be detected by constraint based reasoning. Currently, we are ignoring such induced secondary faults.

The goal of the reasoning process is to find which components could be responsible for an anomalous parameter

value. Following the example of GDE, conflicts are generated which contain components at least one of which must be faulty. Fault hypotheses are then created such that all conflicts are explained. In EDIS conflicts may not accurately reflect the status of the SSME because of modeling inaccuracies and indeterminacies. EDIS will rather post too many conflicts than too few. In GDE components which contribute to a prediction are collected while the prediction is being generated, i.e. during value propagation. Components encountered during propagation are responsible for generating the correct value. EDIS does not propagate values to predict normal values or to find symptoms. Instead, components responsible for symptoms are found after the symptoms, i.e. the anomalous data readings, have been identified.

The reasoning process uses information stored in component models to predict parameter deviations and to verify that a given set of values conforms to the behavior constraints of the relevant components. Input and output are not distinguished since constraints are non-directional. Normal behavior is tacitly assumed. Unknown parameter values or quantities in constraints are assumed to be nominal but such assumptions are made explicit. Note that propagation of normal values is unnecessary in a behavior model describing only deviations. Propagation would only conclude that inferred parameter values are also normal, which is assumed anyway. This is a simplification compared to the generic method using quantitative constraints as exemplified by GDE, but it is only useful if normal values for all important parameters are available.

When symptoms are present, EDIS tries to generate all possible consistent situations which can account for the symptoms. EDIS generates "scenarios" which indicate measured and presumed anomalous parameters and those components which are presumed faulty. Scenarios are derived from constraint models. Each component which lies in the casual path leading from a correct value to an anomaly is examined. If enough data are available, all its fundamental constraints can be verified and the component can be judged good or faulty. In general this is not possible.

If only one side of a fundamental constraint is known, an inference can be made about the other quantity. If the first quantity is normal then the second quantity must also be normal unless the component is faulty. A conflict will arise if the second quantity later turns out to be anomalous. This conflict simply states that the component is faulty since one of its fundamental constraints is violated. If, however, the first, i.e. the known, quantity is anomalous then the component is either faulty or the second quantity is corrupted by another

component or both. A binary conflict between the component being faulty and the quantity being corrupted arises.

If none of the quantities in a fundamental constraint are known, i.e. its normative constraints cannot be evaluated because of lack of data, propagation from neighboring components is used to derive possible scenarios. If the neighboring component has a binary conflict, then both possibilities are considered and, possibly, new conflicts are created by fusing local data with propagated data. When propagation leads to inconsistencies the scenario is impossible.

Propagation is also used in the previous case, i.e. when one side of a fundamental constraint is known, in order to examine the validity of scenarios. At the end of analysis one, several, or no scenarios may exist. If none survives our method has failed. We do not think that this is likely, since no particular fault behaviors are assumed. Faults only manifest themselves as violated constraints. If exactly one scenario is generated, it contains the component or components which are faulty. If several scenarios survive propagation and testing, EDIS or the user have to make a choice. At this point specific fault modes or behaviors may be assumed or simply the number of faults can be minimized, or fault probabilities of components can be utilized to discriminate between fault hypotheses.

Currently, the failure propagation mechanism is implemented using reduced detail, i.e. only anomalies in general are propagated instead of detailed information about size and direction of particular parameter deviations. Conflicts are generated by collecting all components encapsulated between two or more correct readings which exhibit at least one anomalous parameter value. Such a method which does not use predictive models yields too many candidate solutions, but the correct solution, i.e. the component which is responsible for the anomaly, is guaranteed to be among the candidates. At this time candidate (or hypothesis) discrimination proceeds under a single fault assumption. Hypotheses which can explain all anomalies are located by tracing "backwards" through the component structure until a root cause is identified. For simplification the algorithm assumes directional causal relations. Simple common faults, such as turbine problems, can be found using this technique.

4.4 Interconnectivity, Functionality, and Processes

Interconnections are determined when the component is instantiated as part of a specific device or system. Primitive components are grouped in categories for purposes of

organization and to get a better understanding of the domain. The SSME primitive components are grouped in categories in Table 3. The interconnections between the components are illustrated graphically in Fig.7 . As can be seen the SSME system is modeled as interconnected complex and primitive components.

Functionality of a primitive component is defined in terms of one or more fundamental processes - fundamental statements which describe relations among primitive parameters. Parameters describe the state of an object. For strict qualitative modeling, parameters take on discrete values such as "high", "medium", and "low". The current prototype of the EDIS domain uses analytic equations to describe processes to avoid any indeterminacy.

As described above, processes are fundamental statements which describe relations among primitive parameters. The seven processes currently defined in the prototype EDIS system are:

1) pGas Turbine
2) pHydraulic Turbine
3) pTurbopumps
4) pTransmit
5) pDuct
6) pValve
7) pPreburner

The simulation methodology used to derive both deviant and normal behavior is described in Section 5.

TABLE 3. GROUPING OF PRIMITIVE COMPONENTS INTO CATEGORIES

Gas Turbines	Hydraulic Turbines	DUCTS	FLUIDS
LPFTT	LPOTT	F101	HY101
HPFTT		F102	HY102
HPOTT		F103	HY103
		F104	HY104
Pumps	Valves	F105	HY105
		F106	HY106
LPFTP	MFV	F107	HY107
LPOTP	MOV	F108	HY108
HPFTP	FPOV	F109	HY109
HPOTP	OPOV	F110	HY110
	CCV	F111	HY111
		O201	OX201
Combustion Chambers		O202	OX202
		O203	OX203
MCC		O204	OX204
		O205	OX205
Controllers	Nozzles	O207	OX207
MCON	NOZ		
Preburners			
OPB			
FPB			

5. IMPLEMENTATION

5.1 Shell

At the beginning of the project several expert system shell products were evaluated. Important selection criteria included power of representation and inference mechanisms, ease of creating a custom user interface, portability between various hardware platforms, and ease of integration with existing and future software components. Shells which contained support for the object-oriented paradigm were preferred. The selected shell also had to run on a personal computer. Our evaluation ranked NEXPERT-Object first and KES second. However, due to budget constraints we selected KES. KES provides backward chaining rules, data driven demons, and class/member (object-oriented) data representations. In addition we purchased a subroutine package from Quinn-Curtis (QC) which contains support for mathematical functions and graphical data presentation. The QC routines were integrated with KES and provide the user interface framework. KES itself was embedded into a C main program which manages the blackboard and dispatches the KES modules. Embedding KES allows the system designer to develop and test EDIS modules as stand-alone KES applications first and subsequently integrate them into EDIS. Modules can also be written in C, but C modules have to implement blackboard communication explicitly.

KES is currently being updated from version 2.5 to version 3.0. The new version contains an extended window-driven developers interface and support for relations between data objects. Version 3.0 is available for the Hewlett-Packard workstation and is integrated with X Windows. Version 3.0 has not yet been released for the PC.

A listing of the "C Source Code" is contained in Appendix B and one for the "KES Code" in Appendix C.

5.2 Computer Requirements

We are using KES 3.0 on an HP 9000/319 UNIX workstation and KES 2.5 on a Hewlett-Packard QS 16/S personal computer based on the Intel 80386SX microprocessor. The PC uses the DOS 3.3 operating system. KES does not require a 80386-based PC but it is recommended. A numeric coprocessor is recommended especially to enhance the speed of drawing graphic images. A hard disk drive is required and we used at least 640 KBytes of main memory. The display routines can be adapted to any graphics interface but EGA or VGA is recommended for better results.

5.3 File Structure

There are two C header files "comdef.h" and "ssincl.h". "comdef.h" contains data definitions and function declarations for blackboard communications. "ssincl.h" has to be included in every C file to be compiled. It includes all necessary modules.

The file "ssmain.c" contains the main program, while "bbcomm.c" implements the blackboard communication functions. "embed.c" contains the callback interface routines for calls from the embedded KES system. On the PC "menu1.c", "menu2.c", and "menu3.c" contain user interface routines.

On the PC the system can be compiled with the Microsoft C compiler using a "Large" memory model and a stack size of 4000. Linking must include ssmain (which includes header), bbcomm, embed, the "menu?" user interface files, and the Quinn-Curtis files segruah (an adapted version of segraph), worlddr, asyncxx, and hpplot. Care must be taken that the include files for the Quinn-Curtis files can be found by the linker. You may need to use the "I" option of the linker. If you have added modules (knowledge sources) to the system written in C these must also be included.

KES modules must be parsed with the KES compiler. Compiled KES modules must reside in the same directory as the executable "ssmain.exe". The KES module "straty.kb" has to exist; it represents the strategy module which schedules all tasks. Other KES modules currently in use are "freadr.kb" which reads simulated test and comparison data from files, "anomal.kb" which detects and classifies data anomalies, and "diagn2.kb" which attempts to find the fault causing the detected anomalies. Figure 6 illustrates the class hierarchy used to define the engine model. Figure 7 depicts the reasoning model of the preliminary qualitative model.

The existence of both KES and C knowledge sources has to be announced to the system. Enter a function call to "initKS" or "initKSC" in the file "header.c" similar to the ones there. You will also have to make sure that a task is scheduled which uses the new knowledge source. To change task scheduling edit and re-parse the "straty.kb" KES knowledge base. Make sure that the task will correctly identify the knowledge source to use as defined in the "header.c" file. There has to be an "EXIT" task on the blackboard or the program will never terminate.

Data files contain configuration and test data. The file "gconf.dat" lists the generic configuration, i.e. components and interconnections. Data in "sconf.dat" contains the specific

configuration, e.g. turbo machinery serial numbers. Data in "dvarlm.dat" specify how far parameter values may deviate from the comparison data before they are considered anomalous. "tdata.dat" and "cdata.dat" contain simulated test and comparison data in a format directly readable by the KES "freadr.kb" module. On the PC, the user is prompted for these last two file names, all other names are hard-coded in module "freadr.kb".

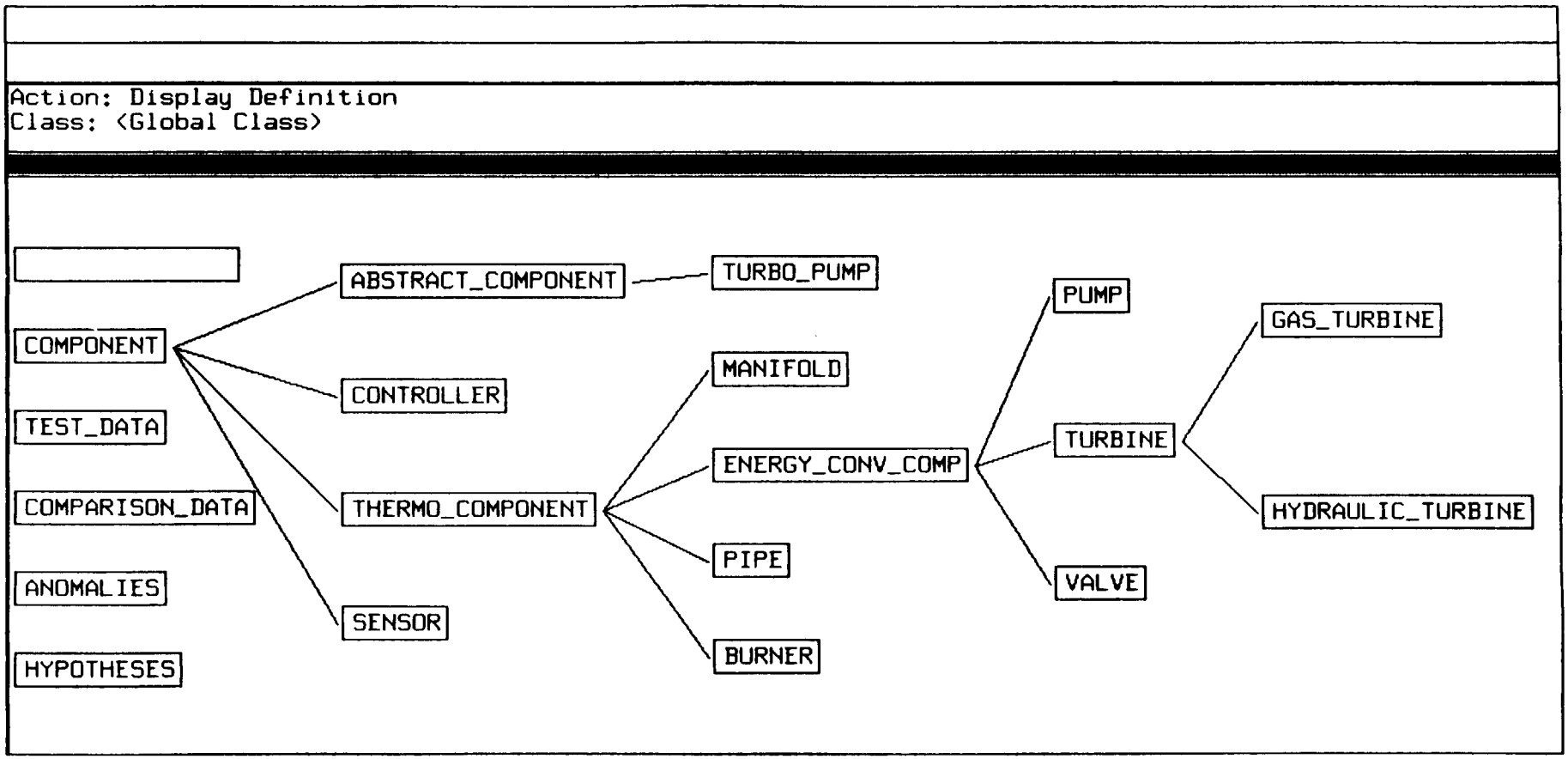


Figure 6. EDIS Component Classification.

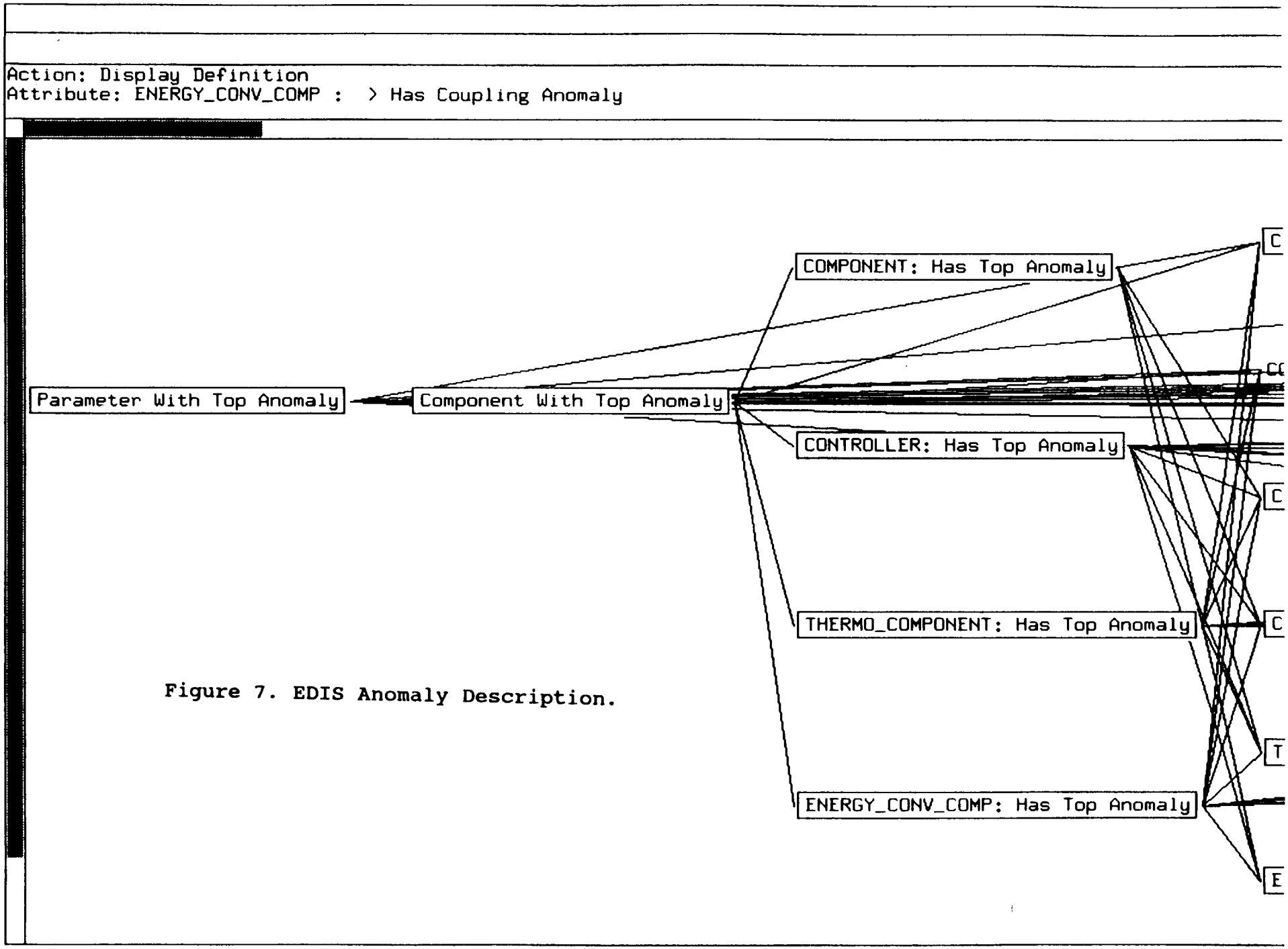
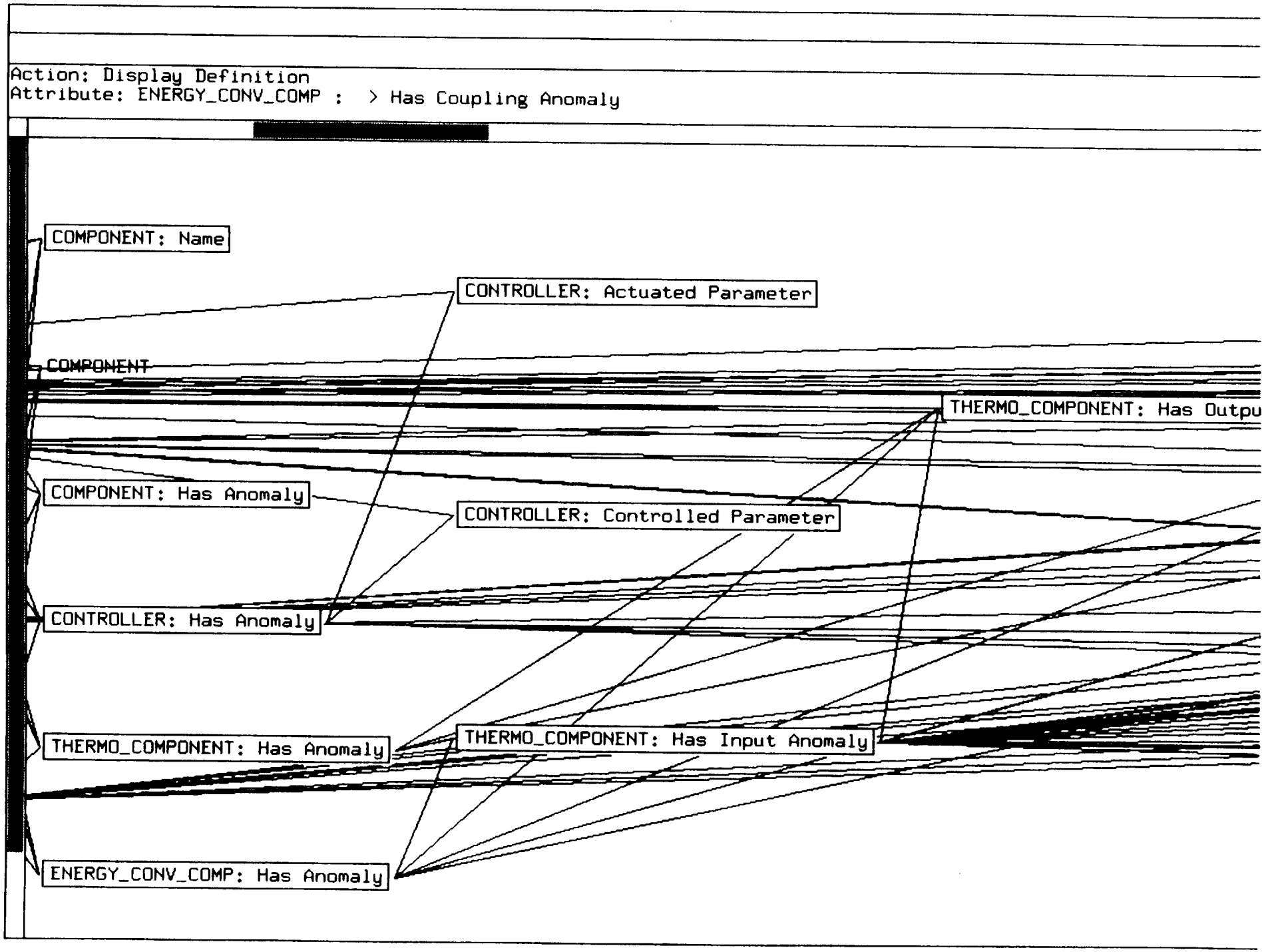
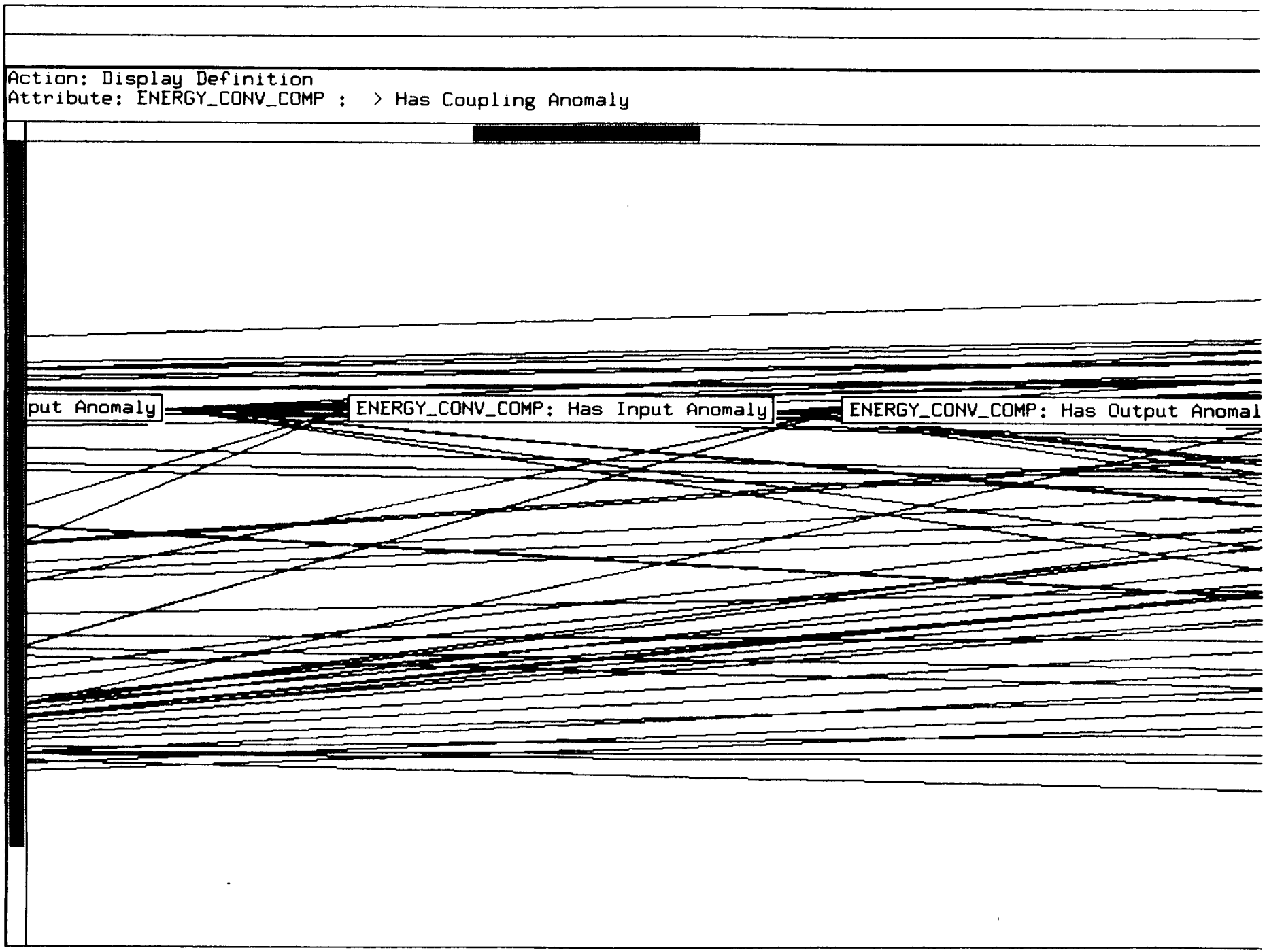


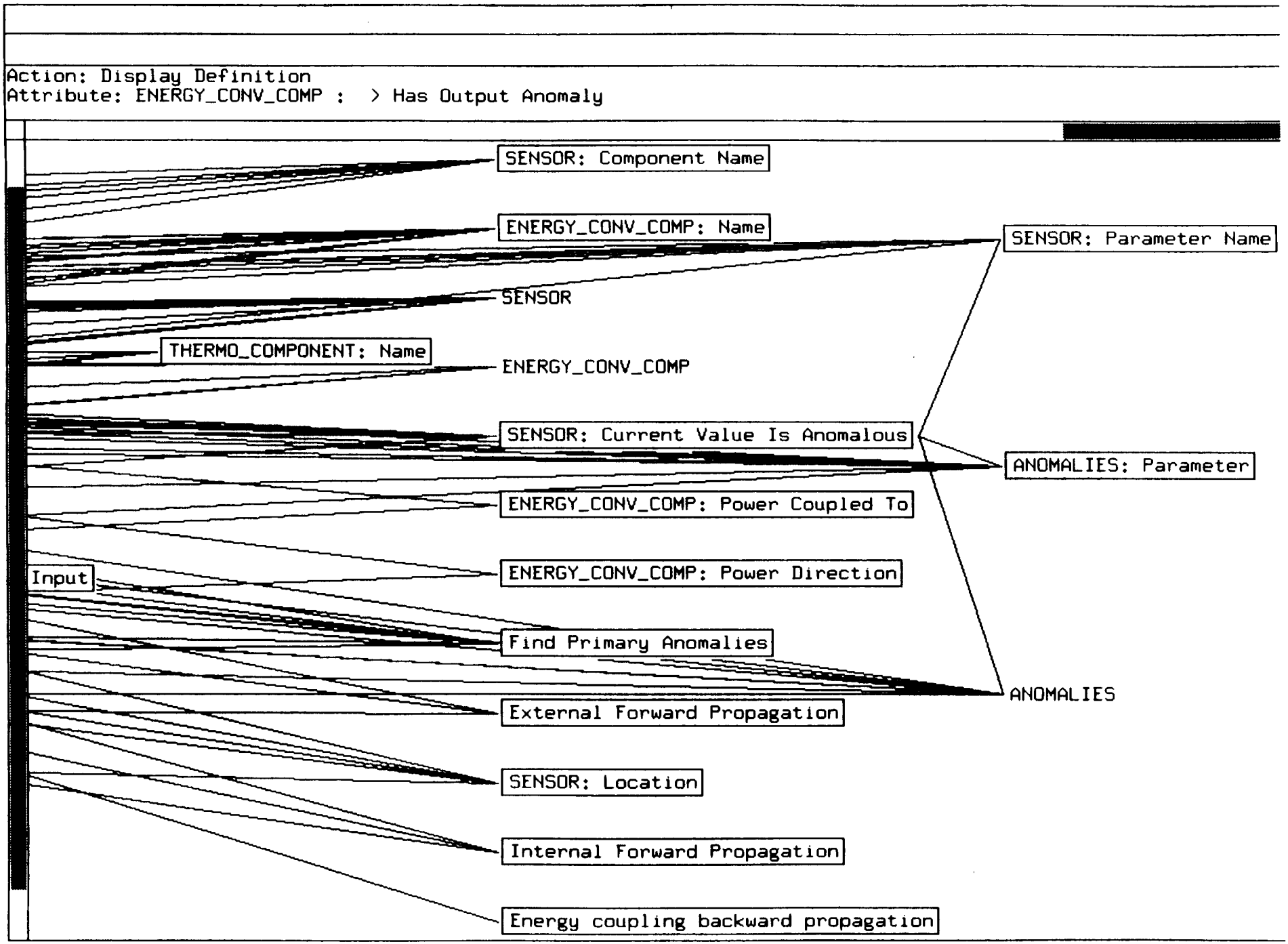
Figure 7. EDIS Anomaly Description.


Action: Display Definition
Attribute: ENERGY_CONV_COMP : > Has Coupling Anomaly
COMPONENT: Name
CONTROLLER: Actuated Parameter
COMPONENT
THERMO_COMPONENT: Has Outpu
COMPONENT: Has Anomaly
CONTROLLER: Controlled Parameter
CONTROLLER: Has Anomaly
THERMO_COMPONENT: Has Anomaly
THERMO_COMPONENT: Has Input Anomaly
ENERGY_CONV_COMP: Has Anomaly


Action: Display Definition
Attribute: ENERGY_CONV_COMP : > Has Coupling Anomaly
put Anomaly
ENERGY_CONV_COMP: Has Input Anomaly
ENERGY_CONV_COMP: Has Output Anomal

Action: Display Definition
Attribute: ENERGY_CONV_COMP : > Has Output Anomaly
ENERGY_CONV_COMP: Has Coupling Anomaly

ORIGINAL PAGE IS
OF POOR QUALITY


Action: Display Definition
Attribute: ENERGY_CONV_COMP : > Has Output Anomaly
SENSOR: Component Name
ENERGY_CONV_COMP: Name
SENSOR
THERMO_COMPONENT: Name
ENERGY_CONV_COMP
SENSOR: Current Value Is Anomalous
ENERGY_CONV_COMP: Power Coupled To
ENERGY_CONV_COMP: Power Direction
Input
Find Primary Anomalies
External Forward Propagation
SENSOR: Location
Internal Forward Propagation
Energy coupling backward propagation
SENSOR: Parameter Name
ANOMALIES: Parameter
ANOMALIES

6. CASE STUDIES

During the course of this investigation interviews were conducted with various experts on SSME engine diagnostics. Several of these interviews are summarized in Appendix C. From these interviews, several special cases of anomalous engine performance were determined and the logic surrounding the diagnosis of these problems investigated. These special cases were then developed into a form for inclusion in EDIS.

6.1 Turbomachinery Malfunctions

Due to the importance of many turbomachinery components in the SSME performance, turbomachinery malfunctions are a common cause for anomalous engine behavior.

High Pressure Fuel Turbopump (HPFTP) Static Seal Leak - In this case there is a leakage of gas past the static seal into the hot gas manifold. This leakage causes a loss in turbine power which, in turn, produces the following effects:

* reduction in turbopump shaft speed
* reduction in flow rate exiting turbine
* reduction in turbopump discharge pressure

The decreased flow rate is sensed by the controller which causes the fuel preburner oxidizer valve to open. This, in turn, increases the preburner oxidizer flow rate.

Under these conditions, the turbopump must do more work for the same power output and the tubrine discharge temperature goes up. If the temperature goes too high, the SSME will shut down.

Obstruction in Inlet Duct to Low Pressure Fuel Turbopump (LPFT) Turbine - An obstruction in the inlet duct to the LPFT turbine by some foreign object (fractured seals, fracture of mozzle vane, glass beads, etc.) causes a loss of energy available to the turbine. This, is turn, results in decreased turbine power and subsequently a:

* reduction in LPFTP shaft speed
* reduction in pump output flow rate
* reduction in pump discharge pressure

The controller senses the increased HPFTP demand and increases the fuel preburner oxidizer flow. The HPFTP can possibly cavitate causing excessive turbine discharge temperatures.

Power Loss in LPFTP Due to Fracture of Stator Vane - As before, a loss of LPFTP turbine power causes a:

* reduction in pump shaft speed
* reduction in pump output flow
* reduction in pump discharge pressure

The controller senses the increased HPFTP demand and increases the fuel preburner oxidizer flow. In the event of cavitation, turbine discharge temperature increases.

6.2 Fuel and Oxidizer Leaks

Another common source of anomalous SSME firing data are fuel and oxidizer leaks in ducts, manifolds, and cooling chambers.

Fuel Leak in the MCC - A drop in the MCC coolant discharge pressure suggests a possiable anomaly. A check in the coolant discharge temperature reveals a concurrent drop suggesting a decreased resistance and increased flow rate through the MCC. The LPFT speed is lower than normal due to the decreased MCC discharged pressure. The MCC coolant flow rate reveals an increased value. These parameters suggest a leak in the MCC coolant tubes.

These anomalies have been investigated and converted into a form for inclusion into EDIS. They are only preliminary and represent the manner in which EDIS will perform diagnoses. Other anomalies and reasoning will be added to EDIS to make it more comprehensive.

7. DISCUSSION

We have developed an architecture and a qualitative reasoning mechanism for reviewing SSME test data and diagnosing SSME faults based on data anomalies. The modular architecture developed for EDIS facilitates modular software development and coordination of different reasoning paradigms. The fault diagnosis methodology presented combines high degrees of fault coverage, domain generality, and domain knowledge. Fault coverage is achieved through constraint-based behavior models and avoidance of fault assumptions. Domain generality is derived from using generic component models and separate connectivity descriptions. Domain knowledge is represented by the component models. Additionally, expert experience is stored in heuristic rules. EDIS incorporates and coordinates reasoning based on heuristic expert knowledge, qualitative models, and quantitative models.

Relation to Other Work - The architecture of EDIS is a variant of the now widely used "blackboard" architecture which was made famous by the HEARSAY project [35]. The blackboard architecture facilitates incremental system development, controlled module interaction, and explicit storage of data and inference results. The reasoning architecture used by EDIS combines qualitative and quantitative reasoning at the hypotheses level which affords more seamless integration than was possible before.

Relevant comparable approaches to diagnosing engineering devices have been introduced and discussed at the beginning of this report. EDIS uses a constraint-based representation for device behavior similar to the one proposed by Davis [13] but adopts a qualitative formulation for the constraints as introduced by de Kleer [24]. EDIS works with models of correct behavior only which has been publicized by Davis and de Kleer (GDE) [20]. We had to adapt the reasoning mechanisms of GDE for the SSME where component models are too weak to propagate values unambiguously. Also, EDIS can reason about possible scenarios based on incomplete information while GDE is silent when no more data can be acquired. EDIS does not resort to pure trial an error constraint suspension but uses constraints on parameters as guidance.

Hudlicka and Lessor [36] have developed a problem-solving system for simulating and diagnosing aircraft behavior which also incorporates and integrates qualitative and quantitative reasoning into a causal model of a complex dynamic system. Their system requires an explicit causal model which defines influences of components on forces and of forces on flight characteristics. The causal model is valid for a specific

configuration. EDIS attempts to reason from component models and interconnectivity information. EDIS can easily be adapted to changes in configuration. The option of creating an explicit causal model from the component and constraint-based model to facilitate diagnosis later may be explored in the future. Their causal model is directional and contains no feedback or cycles and does not describe component behavior by itself as compared to component constraints in EDIS. Their quantitative model, like EDIS, uses simplified linearized equations which are defined for a number of operating conditions. Diagnosis does not start at the detailed level of sensor anomalies used in EDIS but when an alarm at system level is received from a separate diagnostic system. The problem of dealing with multiple faults is therefore simplified because individual fault notices are assumed to be received. In short, the approach taken by EDIS appears to model systems at a deeper level.

Sussman and Steele have developed a general framework for reasoning with non-directional constraints [37]. Their approach uses "equivalence slices" to represent several different views of one set of components and avoids solving simultaneous symbolic equations. Each view contributes different pieces of the final analysis. A slice can also represent concisely the behavior of several components and thus support hierarchical composition. CONSTRAINTS manipulates exact numeric constraints in contrast to EDIS. CONSTRAINTS is based on EL developed by Stallman and Sussman [38] which introduced the method of "propagation of constraints" to analyze electrical circuits. EL dealt with components which display different behaviors in different states by assuming states and retracting inferences when an assumed state lead to an inconsistency. EL retains dependency information to identify the inferences to retract. EDIS must find all possible situations, i.e. all possible combinations of component states, which are consistent with the data. EDIS uses dependency information not to retract facts but to assign blame to faulty components.

De Kleer applied constraint-based reasoning to qualitative analysis of physical systems, in particular to electric circuits [39]. In his EQUAL, system component behavior is expressed by qualitative equations called "confluences". EQUAL, like EDIS, reasons about incremental changes from steady-state operating points. The EDIS constraint "A p B" is completely equivalent to de Kleer's formulation "dA = dB". Both constraints indicate that an increase (or decrease) in A will lead to an increase (or decrease) in B. EDIS is limited to first order constraints while de Kleer's qualitative calculus includes higher order derivatives. EDIS does not mention steady-state constraints because it assumes equilibrium unless otherwise indicated.

EDIS captures causality in terms of function instead of behavior, cmp. EQUAL. Functional causality leads to the distinction between fundamental and normative constraints. The primary function of a component gives rise to fundamental constraints which are only violated in case of a component fault, i.e. when component function is compromised. Given a fundamental constraint, the constrained quantities can be related to measurable quantities using normative constraints and auxiliary relations. Normative constraints in this sense "cause" normative constraints. Functional causality is non-directional within constraints, however. For example, the primary function of a pump is the conversion of mechanical energy into fluid energy. A fundamental constraint of the pump model expresses energy conservation across this conversion. Mechanical power is characterized via torque and radial velocity; fluid power is characterized by fluid energy (and thus pressure difference) and fluid velocity. The normative constraint that torque is inversely proportional to radial velocity is caused by the fundamental constraint which forces their product to be constant.

De Kleer also shows how teleological analysis can lead to an understanding of the function of individual components with respect to the complete device. EDIS incorporates some causal information by virtue of the arrangement of constraints and pseudo-constants and thus falls in between GDE, which ignores causality, and EQUAL, which explicitly models causality.

REFERENCES

1. J. G. Perry, "An Expert Systems Approach to Turbopump Health Monitoring", AIAA paper 88-3117, 24th Joint Propulsion Conference, Boston, July 1988.

2. U. K. Gupata M. Ali, "LEADER - An Integrated Engine Behavior and Design alyisis Based Real Time Fault Diagnostic Expert System for Shuttle Main Engine", Proceedings of the 2nd International Conference on Industrial Engineering Applications of Artificial Intelligence and Expert Systems, UTSI, Tullahoma, TN, published by Association for Computing Machinery, pp. 135-145, June 1989.

3. W. Dietz, E. Kietch, M. Ali, J. of Neural Networks, 1:1, 1989.

4. H. H. Luce R. L. Govind, "Prediction and Diagnosis of Failure in the SSME High Pressure Fuel Turbopump Using Backpropagation Neural Networks", Proceedings of 1st Health Monitoring Conference for Space Propulsion Systems, Cincinnati, published by University of Cincinnati pp. 218-237, November 1989.

5. B. Whitehead, E. Kiech, and M. Ali, "Rocket Engine Diagnostics Using Neural Networks", AIAA Paper No. 90-1892, AIAA 26th Joint Propulsion Conference, July 16-18, 1990.

6. H. Luce and R. Govind, "Neural Network Pattern Recognizer for Detection of Failure Modes in the SSME", AIAA Paper No. 90-1893, AIAA 26;th Joint Propulsion Conference, July 16-18, 1990.

7. A. M. Norman M. Taniguchi, "Development of an Advanced Failure Detection Algorithm for the SSME", AIAA paper 88-3408, 24th Joint Propulsion Conference, Boston, July 1988.

8. Bruce K. Walker and Eric T. Baumgartner, "Comparison of Nonlinear Smoothers and Nonlinear Estimatons for Rocket Engine Health Monitoring", AIAA Paper 90-1891, AIAA 26th Joint Propulsion Conference, July 16-18, 1990.

9. M. Ali and U. Gupta, "An Expert System for Fault and Diagnosis in a Space Shuttle Main Engine", AIAA Paper No. 90-1890, AIAA 26th Joint Propulsion Conference, July 16-18, 1990.

10. G. Biswas, W. J. Hagins, and K. A. Debelak,

"Qualitative Modeling in Engineering Applications," Proc. 1989 IEEE International Conference on Systems, Man, and Cybernetics, Cambridge, Massachusetts, November 1989, pp. 997-1002.

11. K. D. Forbus, "Intelligent Computer-Aided Engineering", AI Magazine, Vol.9, No.3, pp. 23-26, 1988.

12. R. Milne, "Strategies for Diagnosis", IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-17, pp. 333-339, 1987.

13. R. Davis, "Diagnostic Reasoning Based on Structure and Behavior", Artificial Intelligence, Vol. 24, No. 1-3, pp. 347-410, 1984.

14. J. Hogge, "Compiling Plan Operators from Domains Expressed in Qualitative Process Theory", Proc. Sixth AAAI, Seattle, WA, pp. 229-233, 1987.

15. Genesereth, M.R., "The Use of Design Descriptions in Automated Diagnosis", Artificial Intelligence, Vol. 24, No. 1-3, pp. 411-436, Dec. 1984.

16. D. G. Bobrow, (ed.), Qualitative Reasoning about Physical Systems, Cambridge, MA: MIT Press, 1985.

17. Randall Davis, W. Hamscher, "Model-based Reasoning: Troubleshooting", in Shrobe, E.H., (ed.), Exploring Artificial Intelligence, Morgan Kaufmann, California, pp. 297-346, 1988.

18. Frank Pipitone, "The FIS Electronic Troubleshooting System", IEEE Computer, 1986, pp. 68-76.

19. R.R. Cantone, F.J. Pipitone, and W.B. Lander, "Model-Based Probabilistic Reasoning for Electronic Troubleshooting", IJCAI-8, 1983, pp. 207-211.

20. Johan de Kleer, Brian C. Williams, "Diagnosing Multiple Faults",Artificial Intelligence, Vol. 32, No. 1, 1987, pp. 97-130.

21. Raymond Reiter, "A Theory of Dignosis from First Principle", Artificial Intelligence, Vol. 32, No. 1, 1987, pp. 57-95.

22. Rocketdyne, Space Shuttle Main Engine, SSME Description and Operation, E41000/RSS-8559-1-1-1, September 1, 1983.

23. Rocketdyne, RSS-8598-1, Vol. 1: Space Shuttle Main Engine Performance Prediction and Data Reduction Model Description.

ORIGINAL PAGE IS
OF POOR QUALITY

24. Johan DeKleer and John Seely Brown, "A Qualitative Physics Based on Confluences", in D. G. Bobrow, (ed.), Qualitative Reasoning about Physical Systems. Cambridge, MA: MIT Press, 1985, pp. 7-83.

25. K. D. Forbus, "Qualitative Process Theory," .ul Artificial Intelligence, Vol. 24, No. 1-3, pp. 85-168, Dec. 1984.

26. T. Govindaraj, "Qualitative Approximation Methodology for Modeling and Simulation of Large Dynamic Systems: Applications to a Marine Power Plant," IEEE Trans. SMC, Vol. SMC-17, No. 6, Nov./Dec. 1987, pp. 937-955.

27. V. Sembugamoorthy and B. Chandrasekaran, "Functional Representation of Devices and Compilation of Diagnostic Pr Problem Solving Systems", in Experience, Memory, and Reasoning, J. Klodner and C. Riesbach Editors, Laurence Erlbaum Press, pp. 47-73, 1986.

28. T. Bylander, "A Critique of Qualitative Simulation from a Consolidation Viewpoint", IEEE Trans. Syst. Man Cybern., Vol. 18, No. 2, pp. 252-263, Mar.-Apr. 1988.

29. Jane T. Malin, Nick Lance, "Processes in Construction of Failure Management Expert Systems from Device Design Information", IEEE Trans. SMC, Vol. SMC-17, No. 6, Nov./Dec. 1987, pp. 956-967.

30. Pamela Fink, "Control and Integration of Diverse Knowledge in a Diagnostic Expert System", Proc. IJCAI-85, pp. 426-431, 1985.

31. M.J. Pazzani and A.F. Brindle, "Integrating Heuristic Rules and Device Models in Fault Diagnosis", IEEE Software, Vol. 3, No. 2, 1986, pp. 49-50.

32. Klaus W. Pflueger, "Hybrid Diagnostic Strategy for an Expert System Controlled Automatic Test System (EXATS)", IEEE AES Magazine, Vol. 4, No. 10, October 1989, pp. 25-30.

33. B. Chandrasekaran, "Towards a Taxonomy of Problem-Solving Types", AI Magazine, Vol. 4, Winter/Spring 1983, pp. 9-17.

34. Jens Rasmussen, "The Role of Hierarchical Knowledge Representation in Decision-Making and System Management", IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-15, No. 2, 1985, pp. 234-243.

35. L.D. Erman, F. Hayes-Roth, V.R. Lesser, D.R. Reddy, "The

ORIGINAL PAGE IS
OF POOR QUALITY

HEARSAY-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty", Comput. Surveys, Vol. 12, June 1980, pp. 213-253.

36. Eva Hudlicka and Kevin Lesser, "Integrating Causal Reasoning at Different Levels of Abstraction", Proc. of ACM Conference on AI Systems, 1988, pp. 157-163.

37. Gerald Jay Sussman and Guy Lewis Steele Jr., "CONSTRAINTS - A Language for Expressing Almost-Hierarchical Descriptions", Artificial Intelligence, Vol. 14, 1980, pp. 1-39.

38. Richard M. Stallman and Gerald J. Sussman, "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis", Artificial Intelligence, Vol. 9, 1977, pp. 135-196.

39. Johan de Kleer, "How Circuits Work", in D. G. Bobrow, (ed.), Qualitative Reasoning about Physical Systems, Cambridge, MA: MIT Press, 1985, pp. 205-280.

ORIGINAL PAGE IS
OF POOR QUALITY

APPENDIX A: SUMMARY of INTERVIEW SESSIONS

ORIGINAL PAGE IS
OF POOR QUALITY

REPORT (NAS8-36955, D.O. 58)
December 14, 1989
by
Martin Hofmann
Thomas Cost
UAH

Current Understanding of the Review and Diagnosis Process

This report documents the current status of the EDIS project.

Events:

Mike Whitley and Gary Lyles were unavailable for two weeks in December but Mike Whitley arranged a meeting with SSME data review experts for us and has left us some documentation. On Dec. 8 we were able to meet with Marc Neely and Lewis Maddox to discuss the basics of the data review process. We were also introduced to Bruce Boulanger and ??? from Martin Marietta who support the numeric SSME performance predication and data reduction models. We will meet with them again to learn more about the numeric SSME models and how they are used in the data review and diagnosis task.

Tasks:

We have made progress in all three aspects of the Project Assessment Task as defined in the Study Orientation Meeting Document of November 2, 1989. (1) The review of applicable literature is continuing; a partial bibliography is attached. (2) SSME propulsion modeling is being addressed in the context of the Power Balance Model and the Digital Transient Model. (3) The engine diagnosis task was described to us by Marc Neely and we have studied documentation provided by Mike and Bruce.

The following is a summary of our current understanding of the data review and diagnostic processes. We will point out where we still lack information, and where we see potential for a successful expert system application. We would appreciate any corrections or improvements you can suggest for us to incorporate. In our next session with Marc we will try to identify those faults and symptoms which we will incorporate into the prototype system. We will study the knowledge which he applies specifically in solving the chosen problems and we will determine the tools and mechanisms our system will have to employ to mimic his reasoning process.

1. Literature review: a partial bibliography is attached.

2. Propulsion System Modeling:

Documents reviewed:

"SSME Model Analysis Procedure for Ground Test Data Review Support".

"Procedure for Implementing the Power Balance".

"Engine and Pump Performance Calculations Used in TIP87BAB" by J. Taylor Hooper.

ORIGINAL PAGE IS OF POOR QUALITY

"SSME PHASE II Power Balance Average Database", Memo 3912-P/10-89 by B. Boulanger, Jan. 19, 1989.

"RSS-8598-1" describing the Space Shuttle Main Engine Performance Prediction and Data Reduction Model and its usage.

"Space Shuttle Main Engine", Part Number RS007001, SSME Description and Operation, by Rocketdyne, E41000, RSS-8559-1-1-1, Sept. 1, 1983.

3. Engine Diagnosis

Data review is based on knowledge of engine configuration, modulation of the control inputs as scheduled for the test, expected data, and measured actual data. Diagnosis is based on knowledge of how the engine works (in normal and fault situations), how the engine is controlled, and what behavior to expect. Data are analyzed mostly qualitatively and comparatively. Absolute values are less important except when absolute limits are exceeded (e.g. as defined in the interface control document). For example, the DTM returns only relative data. Fault detection is triggered by unexpected levels (relative to the expected levels) of parameters and by phenomena in the data, e.g. steps, spikes, undershoot, overshoot, etc. Data are inspected first for the whole phase and, if a problem is suspected, in more detail, i.e. a data segment is enlarged.

Comparison data:

Digital data are analyzed in three segments: startup, main stage, and shutdown. For each phase comparison data are prepared in advance. Comparison material is derived from several sources. Note: even the control of the engine is fairly constant, e.g. valve openings, etc. However, the test objectives may require changes. In one example, it appeared as if the expert at first treated an observed deviation as a possible problem and he explained it subsequently as being the result of non-standard control as required by a test objective. Thus, instead of generating new specific reference curves for this particular test, a nominal reference was used and expectations for differences were created mentally. Differences can be predicted as relative changes, reference curves would have to be absolute values.

a) 2 sigma limits compiled from all previous tests.

b) Data from (one or two) previous tests with a similarly configured engine and similar test objectives.

c) Absolute (static or generic) limits from the interface control document.

d) Known changes to the engine configuration.

e) Test objectives.

ORIGINAL PAGE IS OF POOR QUALITY

f) PBM predictions (?)

The above mentioned sources define expectations for the actual data and for normal, possible deviations from the expected data. When deviations are observed they may be explained as normal engine variations, effects of wear, effects of replacing components, effects of other engine changes, effects of changed engine control, effects of changed instrumentation, or faults. Faults can be instrumentation faults, data faults, and engine faults.

Fault verification:

If a fault is suspected three hypotheses are tested in order:

i) incorrect sensor readings

ii) incorrect data processing

iii) engine faults

i) Sensor problems can be identified through inspection of raw sensor data. Often sensors are redundantly implemented and can be checked directly against each other. Other times data validity can be checked through dependent data at related sensors. Sometimes instrumentation experts can help to identify or rule out sensor fault modes and fault possibilities, e.g. some sensors cannot read negative values. (We will need more detailed information on types, location, and operation of sensors.) Also, there may be known sensor problems which can explain differences between data. Sensor problems may exist in comparison or test data.

ii) For now we assume correct processing.

iii) Diagnosis of the engine is necessary, see below.

Engine diagnosis:

Starting from a data anomaly the faulty behavior of the engine is reconstructed. Then the cause of the faulty behavior has to be determined. Faulty behavior can be explained from a qualitative understanding/simulation of the corresponding engine parts. Causes for faulty behavior are hypothesized by the expert (based on experience?). Hypotheses are tested with the help of numeric models. For example, the DTM can verify an incomplete ignition hypothesis. Note: "incomplete ignition" may be the cause for some data anomalies but it represents faulty behavior and not a satisfactory diagnosis. However, it is not directly observable and "closer" to a physical or functional cause. Numerical models ONLY simulate behaviors and behavior interaction; an expert has to postulate the underlying defect(s). Therefore, fault hypotheses have to be characterized by fault models for the responsible component. Fault models translate faults into fault behavior. We are told that sometimes no unambiguous explanation for a fault behavior can be found.

In some cases analog data will be requested from the analog data review to gain more information about engine behavior in a given interval where a problem is suspected. Analog data can identify

ORIGINAL
OF POOR QU

imbalance in rotors and ball wear, for example. Also, some data are available from test stand instrumentation, e.g. total fuel and LOX flows.

Engine decomposition into subsystems:

In some situations the behavior of only parts of the engine have to be considered, for example each turbopump, the fuel and the LOX systems.

Qualitative model

The expert uses a mental qualitative model of the structure and function of engine components and controllers to predict behavior. The expert can derive the effects of deviations of one parameter on other parameters in qualitative terms. Controllers will often mask faults by compensating for their effects. Sometimes resulting transients can be observed in the data, other times the controllers act too fast. In feedback situations we can look for separate additional effects of members of the feedback loop. If qualitative simulation cannot identify the cause of a problem, it will at least identify all the involved processes. The expert can then select the most likely ones. Of course, causal relations could be derived which store the fault propagation paths generated by the qualitative reasoning process.

Some important aspects of SSME operation:

Preburner operating levels are closed-loop controlled via oxidizer flow rates through modulation of the preburner oxidizer valves.

The main oxidizer valve, main fuel valve, and chamber coolant valve are scheduled as function of the commanded thrust.

The main oxidizer valve and the fuel preburner oxidizer valves are modulated to maintain engine thrust and mixture ration during steady state.

Problems:

Diagnosis is performed with the help of several cooperating human experts, i.e. digital data expert, analog data expert, instrumentation expert, numeric model expert. We will concentrate on diagnosis using digital data.

Data show strong random variation and instrument resolution is limited.

Comparison of test data segments against reference data seems more important than assumed. It may be necessary to supply numerical data comparison algorithms (in addition to current preprocessing). In the short term we will work with characterizations of the data, e.g. "data show a step at time t", " data show 3% undershoot during time interval t1-t2", or data deviate from reference by 10% in time interval t1-t2". The user will be prompted to perform manual/visual validation by inspecting other, related data curves.

Additional information needed:

ORIGINAL PAGE IS OF POOR QUALITY

Documentation on SSME instrumentation.

When are which numerical models run?

How is the data base of prior failures used?

Examples of in-run versus between-run problems.

Possibly some case records of diagnosed faults.

Possibly training material for novice data review personnel.

Qualitative reasoning about the behavior of the SSME as performed by the human expert, e.g. effects of fuel leaks. More detail than what has been observed is needed.

A closer look at the diagnosis process, best in the context of some relevant fault.

Direct access to the case data base and numeric models. How important is it to directly access data? Should the system simply ask the user to do that manually?

Computer networking: can a PC communicate with the IBM mainframe?

Possible Expert System support:

* Selection of reference material, similar to intelligent data base management.

* Generation of inputs and parameters for numerical models.

* Comparison of data: detection of limit violations. In the first phase of this project it may be too complicated to detect more subtle phenomena in the data.

* Qualitative simulation of SSME behavior with user defined aberrations in the data.

* Fault hypothesis generation from observed faulty behavior, based on causal, functional, and structural interdependencies. Assistance in verification, testing, and discrimination of hypotheses.

BIBLIOGRAPHY

Thomas L. Adams, Gregory L. Orr, Carl J. Tollander, "An Artificial Intelligence Approach to Coordinated Fault Diagnosis, Control, and Planning for the Space Station Electrical Power System," Proc. AIAA Space Systems Technology Conference, San Diego, CA, June 1986, pp. 1-10.

D. G. Bobrow, (ed.), Qualitative Reasoning about Physical Systems. Cambridge, MA: MIT Press, 1985.

T. Bylander, "A Critique of Qualitative Simulation from a Consolidation Viewpoint," IEEE Trans. Syst. Man Cybern., Vol. 18, No. 2, pp. 252-263, Mar./Apr. 1988.

Randall Davis, "Diagnostic Reasoning Based on Structure and Behavior," Artificial Intelligence, Vol. 24, No. 1-3, Dec. 1984, pp. 347-410.

Randall Davis, W. Hamscher, "Model-based Reasoning: Troubleshooting," in Shrobe, E.H., (ed.), Exploring Artificial Intelligence, Morgan Kaufmann, California, 1988, pp. 297-346.

Pamela Fink, "Control and Integration of Diverse Knowledge in a Diagnostic Expert System," Proc. IJCAI-85, pp. 426-431, 1985.

K. D. Forbus, "Qualitative Process Theory," Artificial Intelligence, Vol. 24, No. 1-3, pp. 85-168, Dec. 1984.

Genesereth, M.R., "The Use of Design Descriptions in Automated Diagnosis," Artificial Intelligence, vol. 24, no. 1-3, Dec. 1984, pp. 411-436.

Hofmann, M., Caviedes, J., Bourne, J., Beale, G., Brodersen, A., "Building Expert Systems for Repair Domains," Expert Systems, Vol. 3, No. 1, 1986, pp. 4-12.

T. L. Laffey, W. A. Perkins, T. A. Nguyen, "Reasoning about Fault Diagnosis with LES," IEEE-Expert, Vol. 1, No. 1, pp. 13-20, Spring 1986.

Leinweber, D., "Expert Systems in Space," IEEE Expert, Vol. 2, No. 1, Spring 1987, pp. 26-36.

Jane T. Malin, Nick Lance, "Processes in Construction of Failure Management Expert Systems from Device Design Information," IEEE Trans. SMC, Vol. SMC-17, No. 6, Nov./Dec. 1987, pp. 956-967.

R. Milne, "Fault Diagnosis through Responsibilitiy," Proc. IJCAI-85, pp. 423-425, 1985.

Narayanan, N.H., Viswanadham, N., "A Methodology for Knowledge Acquisition and Reasoning in Failure Analysis of Systems," IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-17, No. 2, 1987, pp. 274-288.

Klaus W. Pflueger, "Hybrid Diagnostic Strategy for an Expert System Controlled Automatic Test System (EXATS)", IEEE AES Magazine, Vol. 4, No. 10, October 1989, pp. 25-30.

ORIGINAL PAGE
COLOR

Vezina, J.M., Sterling, L., "A CLIPS Prototype for Autonomous Power System Control," Proc. Fourth Conf. Artificial Intelligence for Space Applications, Huntsville, AL, Nov. 1988, pp. 211-220.

Wang, C., Zeanah, H., Andersen, A., Patrick, C., "Automatic Detection of Electric Power Troubles," NASA Technical Memorandum TM-86593, Marshall Space Flight Center, AL, 1987.

Wang, C., Zeanah, H., Anderson, A., Patrick, C., Brady, M., Ford, D., "Automatic Detection of Electric Power Troubles," Proc. Fourth Conf. Artificial Intelligence for Space Applications, Huntsville, AL, Nov. 1988, pp. 125-130.

Weeks, D.J., "Artificial Intelligence Approaches in Space Power Systems Automation at Marshall Space Flight Center," Proc. First Internat. Conf. Industrial & Engineering Applications of Artificial Intelligence & Expert Systems (IEA/AIE-88), Tullahoma, TN, June 1988, pp.361-366.

ORIGINAL PAGE IS
OF POOR QUALITY

REPORT (NAS8-36955, D.O. 58)
January 25, 1990
by
Martin Hofmann
Thomas Cost
UAH

Support of the Data Review Process Provided by Numeric Engine Models

Events:

On January 17, 1990 Tom Cost and Martin Hofmann met with Bruce Boulanger and Brian Piekarski from Martin Marietta, who maintain the numeric SSME performance predication and data reduction models. We learned what types of analysis they perform in support of the data review process.

Knowledge Gathered:

Martin Marietta: There are two groups from Martin Marietta involved in the engine tests. One inspects raw measured data, compares these data with data from the previous test (like Marc et al.), and archives the data in a data base. An anomalies data base is kept for raw measured data.

The other group (Bruce & Brian) uses averaged data (from 50 samples per second to one-second averages) for steady-state data sections. Transients are ignored. The PBM, too, works with 1-second average data. 1-second average data are kept in a data base on the IBM. (B&B are supervised by John Butas from NASA.)

NASA itself (Chris Singer?) maintains a data base of test data from which mean and standard deviation values are derived, i.e. the "2-sigma" limits. While B&B work on the IBM, the 2-sigma data are kept on the Perkin-Elmer. The IBM can be accessed via a modem and KERMIT from a PC.

Use of Models: The engine models (mainly the PBM, also the DTM) are used for two main purposes. a) Engine performance evaluation: The group from Martin Marietta provides a second opinion versus Rocketdyne's evaluation of engine performance after each test. b) Support for the data review process as performed by Marc Neely et al., i.e. on a system level. (It appears that Marc etc., besides evaluating the digital data, also coordinate the evaluation of test data by specialists, e.g. numeric model specialists, turbo machinery experts, sensor experts, analog data specialists, etc.)

a) Engine performance evaluation: The main goal is to verify that the engine hardware is performing adequately. The hardware is either being readied for flight or new designs are tested. The PBM is run in the "data reduction mode". Results are presented in comparison with 3 or 4 previous tests. For example, a flow rate at 104% power at steady-state is plotted versus previous test results. Some data patterns indicate problems. For example, a steady decline in flow rate may indicate a leak, or degraded trust may indicate loss of combustion efficiency, measurement problems, or may reflect hardware changes.

Engine components are evaluated separately (mostly pumps) via

ORIGINAL PAGE IS
OF POOR QUALITY

efficiency factors. Efficiency factors are calculated by varying component parameters until the model predictions approximate the measured data. Efficiency factors are used to characterize hardware components. Both individual low efficiency as well as disparate efficiencies of fuel and LOX pumps can lead to engine problems. Excessively low efficiency will manifest itself in other measured parameters, e.g. high pump discharge temps, so that model-based analysis is not always necessary. For subsequent tests the calculated efficiency factors will be used to predict overall engine performance. Since pumps are interchangeable individual performance histories are kept for them.

b) Data Review Support: Generally no pretest predictions are made except for actual flights, but sometimes KF and C2 factors are estimated based on the last test. KF and C2 calibrate fuel flow and mixture ratio measurements. The model can be used to test fault hypotheses for leaks and flow resistances. The model can simulate leaks and increased flow resistance and its results will be interpreted as deviations (or "deltas") from the nominal parameters. The deltas will be indicated in a schematic of the engine configurations. Also, plots of nominal parameter values versus anomalous values for various power levels may be produced.

Main Model Parameters:

INPUT:

LOX flow rate \
fuel flow rate / from facility meters
calculated thrust
ISP chart (specific impulse: thrust/flow rate)
nominal mixture ratio

OUTPUT:

High pressure (HP) turbine discharge temps	(2)
HP turbine pressures	(2)
HP pump fuel inlet pressure	(1)
LP fuel turbine inlet temp	(1)
preburner chamber pressures	(2)
HP oxidizer pump suction specific speed	(1)

Tasks Performed:

We have completed the project assessment phase and are starting the project specification phase.

Possible additional benefits from an expert system:

The expert system could exercise the numeric models in support of Marc Neely et al. in standard situations without intervention by Martin Marietta personnel. This would permit better integration and speedier execution of the review process. The expert system would thus capture expertise necessary to run the numeric engine models.

APPENDIX B: LISTING of EDIS C-SOURCE CODE

ORIGINAL
OF POOR QUALITY

```
#include "ssincl.h"

int i,j,k,l;
int success;
long int v_time = 0;                    /* virtual time */
class_ptr blackboard = NULL;        /* define the unique blackboard */
int bb_size = 0;                    /* number of classes on the blackboard */
member_ptr first_task;              /* head ptr for task list */
member_ptr current_task;
member_ptr temp_mem;
attr_ptr temp_attr;
attr_ptr *app;
KES_command_type command_type;
KES_attribute_type temp_KES_attr;
KES_atr_seq_type temp_KES_attr_seq;
KES_class_type temp_KES_class, temp_KES_class2;
KES_member_type temp_KES_mem;
KES_value_type temp_KES_value;
KES_class_value_type temp_KES_cv;
int (*fct_ptr) ();

main()
{ int max_prio, num_attrs;
  char *t_name, file_name[LINE_LENGTH], temp_str_arr[LINE_LENGTH];
#ifdef MSDOS
  char temp33[LINE_LENGTH];
#endif
  char *KS_name, *KS_fct, *KS_dir, *C_name, *temp_str, *class_name;
  char *attr_name, *mem_name, *temp_str2;
  char KS_io_list[LINE_LENGTH], t_string[LINE_LENGTH];
  member_ptr t_mem;
  attr_val_type t_aval;

  current_task = NULL;
  first_task = NULL;

  initialize(&blackboard);
#ifdef MSDOS
  initialize_windows();
#endif

  /* basic loop = task dispatcher */
  /* THIS FINDS TASKS ON THE BB! */
  for (;;) {
    /* find task on BB with highest priority */
    first_task = bb_find_mem(blackboard, "TASK"); /* get first task */
    current_task = first_task;
    max_prio = atoi(find_attr_sval(current_task, "priority"));
    current_task = current_task->fwd;
    while (current_task != NULL) {
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
      temp_attr = find_attr(current_task->attributes, "priority");
      if (max_prio < atoi(temp_attr->attr_value.sval)) {
  first_task = current_task;
  max_prio = atoi(temp_attr->attr_value.sval);
      }
      current_task = current_task->fwd;
    };
    /* now first_task holds the task with highest priority */
    /* assume there alwas is one: there should always be a
       task to run the strategist left on the BB.  To exit
       use an explicit EXIT task!                          */

    /* Check for EXIT task */
    t_name = find_attr_sval(first_task, "task name");
    if (strcmp(t_name, "EXIT") == 0) {
      clean_up();
      exit(0); /************** PROGRAM EXIT ************************/
    };
    /* otherwise do task: retrieve KS */

    /* For KES task: KS_load, BB_read, KS_execute, BB_write, KS_unload */
    /* For C program: run program with parameter pointer. */
    /* retrieving the KS and task name */
    v_time++;
    KS_name = find_attr_sval(first_task, "knowledge source");
    KS_fct = find_attr_sval(first_task, "task name");
    temp_mem = find_mem(bb_find_mem(blackboard, "KS"), KS_name);
    if (strcmp(find_attr_sval(temp_mem, "KS kind"), "KES") == 0) {
      /* KES knowledge source */
      strcpy(file_name, find_attr_sval(temp_mem, "exec file name"));
      strcat(file_name, EXTENS);
#ifdef HP-UX
      printf("Executing KES task %s using file %s.\n", t_name, file_name);
#endif
#ifdef MSDOS
      sprintf(temp33, "Executing KES task %s using file %s.\n", t_name, file_name);
  init_message_window();
      display_as_message(temp33);
#endif
      /* KS load  */
      if (KES_ld_kb(file_name, 60000L) != KES_success_c)/* (1 != 1)*/ {
  printf("Cannot load KES file %s.\n", file_name);
      }
      else {
  /* BB read */

/* test puts("BB read."); */
  strcpy(KS_io_list, find_attr_sval(temp_mem, "IN"));
  temp_str = KS_io_list;
  while ((temp_str = strtok(temp_str, " ,")) != NULL) {
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
    /* now assert all members */
    t_mem = bb_find_mem(blackboard, temp_str);
    while (t_mem != NULL) {
      class_name = t_mem->class_name;
      /* execute commands like: reassertclass class = class + member */

      temp_KES_class = KES_g_named_class(class_name);
      KES_parse_members(temp_KES_class, t_mem->member_name,
                        KES_false_c, &temp_KES_cv);
      KES_reassertclass(temp_KES_class, KES_true_c,
                        KES_null_class_value_c, temp_KES_cv);
#ifdef HP-UX
/* test */  puts(reassertclass_com_gen(class_name, t_mem->member_name));
#endif
#ifdef MSDOS
/* test */  display_as_message(reassertclass_com_gen(class_name, t_mem->member_name));
#endif
      temp_attr = t_mem->attributes;
      while (temp_attr != NULL) {
        temp_KES_attr = KES_g_named_atr(temp_KES_class, temp_attr->attr_name);
        temp_KES_mem = KES_g_named_member(temp_KES_class, t_mem->member_name);
        KES_parse_value(temp_KES_mem, temp_KES_attr, temp_attr->attr_value.sval,
                        &temp_KES_value);
        KES_reassert(temp_KES_mem, temp_KES_attr, temp_KES_value);

        temp_attr = temp_attr->fwd;
      };
      t_mem = t_mem->fwd;
    };
    temp_str = NULL; /* get ready for next call to strtok */
  };
    /* KS execute etc. */
    KS_dir = find_attr_sval(temp_mem, "inference direction");
    if (strcmp(KS_dir, "forward") == 0) {
#ifdef HP-UX
/* test */  puts(reassert_glob_com_gen(find_attr_sval(temp_mem,
                                                      "function"), "true"));
#endif
#ifdef MSDOS
/* test */  display_as_message(reassert_glob_com_gen(find_attr_sval(temp_mem,
                                                      "function"), "true"));
#endif
      temp_KES_attr = KES_g_named_atr(KES_global_class_c,
                                      find_attr_sval(temp_mem,
                                                     "function"));
      KES_parse_value(KES_global_member_c, temp_KES_attr, "true",
                      &temp_KES_value);
      KES_reassert(KES_global_member_c, temp_KES_attr, temp_KES_value);
    }
    else {
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
#ifdef HP-UX
/* test */  puts(obtain_com_gen(find_attr_sval(temp_mem, "function")));
#endif
#ifdef MSDOS
/* test */  display_as_message(obtain_com_gen(find_attr_sval(temp_mem, "function")));
#endif
     KES_obtain_atr(KES_global_member_c,
                    KES_g_named_atr(KES_global_class_c,
                                    find_attr_sval(temp_mem, "function")
                                    ));
    };
    /* BB write */
    strcpy(KS_io_list, find_attr_sval(temp_mem, "OUT"));
    class_name = KS_io_list;
    /* for all classes on the OUT list */
    while ((class_name = strtok(class_name, " ,")) != NULL) {
      /* prepare access to attributes (using level 3) */
      temp_KES_class = KES_g_named_class(class_name);
      /* now step through all class members */
      for (temp_KES_mem = KES_g_next_member(temp_KES_class,
                                            KES_null_member_c);
           temp_KES_mem != KES_null_member_c;
           temp_KES_mem = KES_g_next_member(temp_KES_class,
                                            temp_KES_mem)) {
        /* create a new C member */
        t_mem = new_member();
        strcpy(t_mem->member_name,
               KES_g_member_name(temp_KES_mem));
        t_mem->attributes = NULL;
        t_mem->n_attr = 0;
             strcpy(t_mem->class_name,
              KES_g_class_name(KES_g_member_class(temp_KES_mem)));
        /* the specific class (not a superclass) */
        temp_KES_class2 = KES_g_named_class(t_mem->class_name);
        temp_KES_attr_seq = KES_g_atrs(temp_KES_class2);
        num_attrs = KES_g_num_atrs(temp_KES_attr_seq);
        /* retrieve and add the attributes */
        for (i = 1; i <= num_attrs; ++i) {
          temp_KES_attr = KES_g_nth_atr(temp_KES_attr_seq, i);
          attr_name = KES_g_atr_name(temp_KES_attr);
          if (KES_g_atr_status(temp_KES_mem,temp_KES_attr)
              == KES_known_c) {
            temp_str = KES_d_value(temp_KES_mem, temp_KES_attr);
            /* parse the value string! (get only first value!) */
            /* throw away anything after a "<" character: this
               should be the certainty factor of the first value */
            strcpy(temp_str_arr, temp_str);
            l = strcspn(temp_str_arr, "<");
            temp_str_arr[l] = '\0';
            while ((isalnum(temp_str_arr[strlen(temp_str_arr)-1]) == 0)
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
                && (strlen(temp_str_arr) > 0)) {
            temp_str_arr[strlen(temp_str_arr) - 1] = '\0';};
          while ((isalnum(temp_str_arr[0]) == 0)
                && (strlen(temp_str_arr) > 0)) {
            strcpy(t_string, temp_str_arr+1);
            strcpy(temp_str_arr,t_string);};
          temp_str = temp_str_arr;
          /* now add the attribute */
          add_attr(&(t_mem->attributes), attr_name, temp_str);
          (t_mem->n_attr)++;
          };
        };
        /* finally insert on BB */
        bb_insert(&blackboard, class_name, t_mem);
      };
      class_name = NULL;
    };
    /* KS unload */
  KES_free_kb();
#ifdef MSDOS
  ClearWindow();
#endif
  }
    }
    else {
      /* C program */ /* In header.c: define the C function as a KS */
      /* C programs can manipulate the blackboard directly.  They
   are responsible for keeping it in correct format */

      printf("Executing C task %s.\n", t_name);

      fct_ptr = find_attr_gptr(temp_mem, "executable function");
      success = (*fct_ptr)();
    };
    /* reset task priority to 0 and set attr "done at" to current "<time>" */
    temp_attr = find_attr(first_task->attributes, "priority");
    strcpy(temp_attr->attr_value.sval, "0");
    if ((temp_attr = find_attr(first_task->attributes, "done at")) == NULL) {
      add_attr(&(first_task->attributes), "done at", time_gen());
    }
    else { /* change time of last execution */
      strcpy(temp_attr->attr_value.sval, time_gen());
    }
  };
};


/* initialization: for each knowledge source add a member to
   the knowledge source class on the blackboard; then
   create a task to run the strategist */
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
void initialize(bbp) bb_ptr bbp; {
  char c_tmp[MAXNAM];
  int success;
  /* add knowledge sources to BB: include a file with calls to init_KS */

#include "header.c"

  /* create task to run strategist */
  temp_mem = new_member();
  strcpy(temp_mem->member_name, name_gen("task"));
  temp_mem->n_attr = 4;
  temp_mem->attributes = NULL;
  strcpy(temp_mem->class_name, "TASK");

  /* create attributes */
  /* attrs: task_name = "find_task", KS = "strategist", time = <now>.
     Thus: there must be a knowledge source on the BB with name
     strategist; it must have a function "find_task"; and the
     KS must be either an executable C program or a parsed knowledge
     base. */
  sprintf(c_tmp, "%i", v_time); /* virtual time as char string */
  app = &(temp_mem->attributes);
  add_attr(app, "time", c_tmp);
  add_attr(app, "priority", "100");
  add_attr(app, "task name", "find task");
  add_attr(app, "knowledge source", "strategist");

  /* put it on the blackboard */
  success = bb_insert(bbp, "TASK", temp_mem);
  switch (success) {
  case -1: printf("Cannot initialize the blackboard!\n"); exit(-1);
    break;
  default: printf("Initialized the blackboard.\n");
    break;
  }

  /* Supply the file names for the configuration files */
  /* This could be done in a separate task using user confirmation/changes */
  temp_mem = new_member();
  strcpy(temp_mem->member_name, name_gen("file"));
  temp_mem->n_attr = 3;
  temp_mem->attributes = NULL;
  strcpy(temp_mem->class_name, "FILE");
  /* create attributes */
  app = &(temp_mem->attributes);
  add_attr(app, "Name", "sconf.dat");
  add_attr(app, "Type", "specific configuration");
  add_attr(app, "Comparison Type", "none");
  /* put it on the blackboard */
  success = bb_insert(&blackboard, "FILE", temp_mem);
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
  temp_mem = new_member();
  strcpy(temp_mem->member_name, name_gen("file"));
  temp_mem->n_attr = 3;
  temp_mem->attributes = NULL;
  strcpy(temp_mem->class_name, "FILE");
  /* create attributes */
  app = &(temp_mem->attributes);
  add_attr(app, "Name", "dvarlm.dat");
  add_attr(app, "Type", "variation limits");
  add_attr(app, "Comparison Type", "none");
  /* put it on the blackboard */
  success = bb_insert(&blackboard, "FILE", temp_mem);

  temp_mem = new_member();
  strcpy(temp_mem->member_name, name_gen("file"));
  temp_mem->n_attr = 3;
  temp_mem->attributes = NULL;
  strcpy(temp_mem->class_name, "FILE");
  /* create attributes */
  app = &(temp_mem->attributes);
  add_attr(app, "Name", "gconf.dat");
  add_attr(app, "Type", "general configuration");
  add_attr(app, "Comparison Type", "none");
  /* put it on the blackboard */
  success = bb_insert(&blackboard, "FILE", temp_mem);


};

/* auxiliary functions */
/************************/

/* name_gen generates a unique name derived from a supplied
   string and the virtual time.
*/
char *name_gen(stem)
     char *stem; {
       static char tmp[MAXNAM];
       static long int tag;
       sprintf(tmp, "%s%i", stem, tag++);
       return tmp;
     };

/* time_gen generates a string represntation from the global integer time */
char *time_gen() {
  static char tmp[MAXNAM];
  sprintf(tmp, "%i", v_time);
  return tmp;
};
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
void init_KS(bbp, name, fct, dir, exec, kind, in, out)
     bb_ptr bbp; char *name, *fct, *dir, *exec, *kind, *in, *out; {
       temp_mem = new_member();
       strcpy(temp_mem->member_name, name);
       temp_mem->n_attr = 6;
       temp_mem->attributes = NULL;
       strcpy(temp_mem->class_name, "KS");
       app = &(temp_mem->attributes);
       add_attr(app, "function", fct);
       /* "function" and "inference direction" specify how to run
    a KES module: if direction=forward then 'assert function=true';
    if direction=backward then 'obtain function'.
    */
          /* only one function per KS at this point */
          add_attr(app, "inference direction", dir);
          add_attr(app, "exec file name", exec);
          add_attr(app, "KS kind", kind);
          /* Now define what data are passed between the BB and the KES module
    Specify classes (both BB classes and KES classes!)
    Syntactic limitation: use single word class names. */
          add_attr(app, "IN", in);
          add_attr(app, "OUT", out);
          /* put it on the blackboard */
          success = bb_insert(bbp, "KS", temp_mem);
          if (success == -1) {
             printf("ERROR initializing %s\n",temp_mem->member_name); exit(-1);
          };
};

/* clean_up is called before program exit. */
void clean_up() {
#ifdef MSDOS
    CloseSEGraphics();
#endif
    puts("All tasks on the agenda have been carried out!");
    puts("EXIT");
    };

/* init_KS_C creates a member of class KS for a C function.  It uses
   a pointer to the function as an attribute value! */
void init_KS_C(bbp, name, fct_comment, fct_exec, kind, in, out)
/*     bb_ptr bbp; char *name, *fct_comment; int (*fct_exec)(void *);*/
     bb_ptr bbp; char *name, *fct_comment; int (*fct_exec)();
     char *kind, *in, *out; {
       temp_mem = new_member();
       strcpy(temp_mem->member_name, name);
       temp_mem->n_attr = 5;
       temp_mem->attributes = NULL;
       strcpy(temp_mem->class_name, "KS");
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
        app = &(temp_mem->attributes);
        add_attr(app, "function", fct_comment);
        /* "function" states what the C function does. */
        add_general_attr(app, "executable function", 4, fct_exec);
        add_attr(app, "KS kind", kind);
        /* Now define what data are passed between the BB and the KES module
     Specify classes (both BB classes and KES classes!)
     Syntactic limitation: use single word class names. */
        add_attr(app, "IN", in);
        add_attr(app, "OUT", out);
        /* put it on the blackboard */
        success = bb_insert(bbp, "KS", temp_mem);
        if (success == -1) {
           printf("ERROR initializing %s\n",temp_mem->member_name); exit(-1);
        };
};

/* reassertclass_com_gen generates a command string to add a member
   to a class. */
char *reassertclass_com_gen(class_name, member_name)
     char *class_name, *member_name; {
       static char command_string[LINE_LENGTH];
#ifdef HP-UX
       sprintf(command_string, reassertclass_format,
         class_name, member_name);
#endif
#ifdef MSDOS
       sprintf(command_string, reassertclass_format, class_name,
         class_name, member_name);
#endif
       return command_string;
     };

/* reassert_com_gen generates a command string to change the values of
   and attribute.  The value has to be passed in as a string. */
char *reassert_com_gen(class_name, member_name, attr_name, attr_sval)
     char *class_name, *member_name, *attr_name, *attr_sval; {
       static char command_string[LINE_LENGTH];
       sprintf(command_string, reassert_format, class_name, member_name,
         attr_name, attr_sval);
       return command_string;
     };

/* reassert_str_com_gen generates a command string to change the values of
   and attribute.  The value has to be passed in as a string. The value
   string will be enclosed in quotes to satisfy KES for values of
   KES type "str" */
char *reassert_str_com_gen(class_name, member_name, attr_name, attr_sval)
     char *class_name, *member_name, *attr_name, *attr_sval; {
       static char command_string[LINE_LENGTH];
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
      sprintf(command_string, reassert_str_format, class_name, member_name,
        attr_name, attr_sval);
      return command_string;
    };

/*reassert_glob_com_gen generates a command string to change the values of
  a global attribute. */
char *reassert_glob_com_gen(attr_name, attr_val)
    char *attr_name, *attr_val; {
  static char command_string[LINE_LENGTH];
  sprintf(command_string, reassert_glob_format, attr_name, attr_val);
  return command_string;
};


/* obtain_com_gen generates a command string which obtains the value
   for a global attribute given by attr_name. */
char *obtain_com_gen(attr_name) char *attr_name; {
  static char command_string[LINE_LENGTH];
  sprintf(command_string, obtain_format, attr_name);
  return command_string;
};

/* display_com_gen generates and executes a command string which returns the
   value of an attribute of a class member.  The returned string
   has the format "attribute value <certainty>". */
char *display_com_gen(class_name, member_name, attr_name)
    char *class_name, *member_name, *attr_name; {
      static char command_string[300];
      sprintf(command_string, display_format, class_name, member_name,
        attr_name);
/*      command_string = KES_command(command_string); */
/* not ready to execute, just display */
      puts(command_string);
      return command_string;
    };

#ifdef MSDOS
/* initialize_windows sets up the Quinn Curtis graphics windows */
int initialize_windows() {
  InitSEGraphics(6);
  /* any changes to the default windows go here */
  SetPercentWindow(0.0,0.0,1.0,1.0,2); /* window 2 is full screen */
  return 0;
  };
#endif

#ifdef HP-UX

int greet_user() {
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
      puts("Hello!");
      return 0;
    };

int get_file_name() {
  char *f1 = "tdata.dat";
  char *f2 = "cdata.dat";
  member_ptr temp_mem;

  temp_mem = new_member();
  strcpy(temp_mem->member_name, name_gen("file"));
  temp_mem->n_attr = 3;
  temp_mem->attributes = NULL;
  strcpy(temp_mem->class_name, "FILE");
  /* create attributes */
  app = &(temp_mem->attributes);
  add_attr(app, "Name", f1);
  add_attr(app, "Type", "test data");
  add_attr(app, "Comparison Type", "none");
  /* put it on the blackboard */
  success = bb_insert(&blackboard, "FILE", temp_mem);

  temp_mem = new_member();
  strcpy(temp_mem->member_name, name_gen("file"));
  temp_mem->n_attr = 3;
  temp_mem->attributes = NULL;
  strcpy(temp_mem->class_name, "FILE");
  /* create attributes */
  app = &(temp_mem->attributes);
  add_attr(app, "Name", f2);
  add_attr(app, "Type", "comparison data");
  add_attr(app, "Comparison Type", "previous test");
  /* put it on the blackboard */
  success = bb_insert(&blackboard, "FILE", temp_mem);

  return 0;
};
#endif
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
/* This is the basic blackboard communication module.  It contains
   functions to assert data into KES modules and to retrieve data
   from KES modules into a C program.  The data are described by
   "HEADERS" which define the sharable data types.  Data types must
   be KES classes.  HEADER information must be provided
   for each knowledge base (KES module) in the file "header.c".
   Add information for each knowledge source you provide there.
   (The HEADER information will be put on the blackboard.)
   All data declared OUT
   will be extracted from the KES module and made available on the
   blackboard.  All data declared IN (instances of classes declared IN)
   will be asserted into a KES module before any inferences take place.
   The blackboard is declared in file "comdef.h"; the overall include
   structure is defined in "ssincl.h",
   (A KES module is a parsed KES knowledge base.)
*/

#include "ssincl.h"

/* Family of blackboard FIND functions; they all have the
   blackboard as their first parameter */

/*bb_find_class returns a pointer to the class whose name is given or NULL */
class_ptr bb_find_class(bb, class_name) class_ptr bb; char* class_name; {
  class_ptr running_ptr;
  running_ptr = bb;
  while (running_ptr != NULL)
    if (strcmp(running_ptr->class_name, class_name) == 0) {
      return running_ptr;
    }
    else {
      running_ptr = running_ptr->fwd;
    };
  return NULL;
};

/* bb_find_mems looks for the first member of a given class and returns
   a pointer to it or NULL */
member_ptr bb_find_mem(bb, class_name) class_ptr bb; char* class_name; {
  class_ptr c_ptr;
  member_ptr m_ptr;
  if ((c_ptr = bb_find_class(bb, class_name)) != NULL) {
    if ((m_ptr = c_ptr->members) != NULL) {
      return m_ptr;
    }
  };
  return NULL;
};

/* find_mem returns a member with given name or NULL */
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
member_ptr find_mem(first_mem, member_name) member_ptr first_mem;
     char* member_name; {
  member_ptr m_ptr = first_mem;
  while (m_ptr != NULL) {
    if (strcmp(m_ptr->member_name, member_name) == 0) {
      return m_ptr;
    }
    else {
      m_ptr = m_ptr->fwd;
    }
  };
  return NULL;
};

/* bb_find_val looks for members of a class which have a given value
   for a given attribute or NULL*/
/* FOR NOW: return only the first one found */
/* FOR NOW: assume all values are strings (as returned from KES!) */
member_ptr bb_find_val(bb, class_name, attribute, value)
     class_ptr bb; char *class_name; char *attribute;
                attr_val_type value/*; utype value_type*/; {
  class_ptr c_ptr;
  member_ptr m_ptr;
  attr_ptr a_ptr;
/*  member_ptr result;
  member_ptr aux_ptr;
  int matches; */
  if (c_ptr = bb_find_class(bb, class_name)) {
    for (m_ptr = c_ptr->members; m_ptr != NULL; m_ptr = m_ptr->fwd) {
      /* find attribute */
      a_ptr = find_attr(m_ptr->attributes, attribute);
      if (a_ptr != NULL) {
  /* assume string values !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!*/
  if (strcmp(a_ptr->attr_value.sval,value.sval) == 0) {
    return m_ptr;
  }
      }
    }
  };
  return NULL;
};

/* find_attr returns an pointer to an attribute of a member or NULL */
attr_ptr find_attr(running_ptr, attribute)
            attr_ptr running_ptr; char* attribute; {
  while (running_ptr != NULL)
    if (strcmp(running_ptr->attr_name, attribute) == 0) {
      return running_ptr;
    }
    else {
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
      running_ptr = running_ptr->fwd;
    };
  return NULL;
};

/* find_attr_sval returns the (string) value of an attribute in an
   attribute list or NULL */
char *find_attr_sval(m_ptr, attribute)
     member_ptr m_ptr; char *attribute; {
       attr_ptr a_ptr;
       a_ptr = find_attr(m_ptr->attributes, attribute);
       if (a_ptr != NULL) return a_ptr->attr_value.sval;
       else return NULL;
     };
int find_attr_ival(m_ptr, attribute)
     member_ptr m_ptr; char *attribute; {
       attr_ptr a_ptr;
       a_ptr = find_attr(m_ptr->attributes, attribute);
       if (a_ptr != NULL) return a_ptr->attr_value.ival;
     };
float find_attr_fval(m_ptr, attribute)
     member_ptr m_ptr; char *attribute; {
       attr_ptr a_ptr;
       a_ptr = find_attr(m_ptr->attributes, attribute);
       if (a_ptr != NULL) return a_ptr->attr_value.fval;
     };
void *find_attr_gptr(m_ptr, attribute)
     member_ptr m_ptr; char *attribute; {
       attr_ptr a_ptr;
       a_ptr = find_attr(m_ptr->attributes, attribute);
       if (a_ptr != NULL) return a_ptr->attr_value.gptr;
     };


/* Family of insert/change/delete bb functions */
/* RULE: a member which exists will be modified but never re-inserted.
   specified values will overwrite existing ones, other values remain. */
/* WATCH: since the Blackboard may be modified it is necessary to
   pass a pointer to the BB, instead of the BB pointer itself, e.g.
   the BB is initially empty and we need to change the BB pointer itself!
   Use type bb_ptr! */

/* memory allocation */
class_ptr new_class() {
  return (class_ptr) malloc(sizeof(class_type));
};

member_ptr new_member() {
  return (member_ptr) malloc(sizeof(member_type));
};
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
attr_ptr new_attr() {
  return (attr_ptr) malloc(sizeof(attr_type));
};

/* bb_insert adds new information.  Returns 0 if new member was
   inserted, 1 for update of existing member, -1 otherwise */
int bb_insert(bbp, class_name, n_member)
       bb_ptr bbp; char *class_name; member_ptr n_member; {
  class_ptr c_ptr;
  member_ptr m_ptr;
  attr_ptr a_ptr, a2_ptr;
  int rval = 0;
  if ((c_ptr = bb_find_class(*bbp, class_name)) == NULL) {
    /* new class */
    /*************/
    if (bb_size < MAXBB) {
      c_ptr = new_class();
      /* insert at beginning of bb */
      bb_size += 1;
      if (*bbp != NULL) (*bbp)->bwd = c_ptr;
      c_ptr->fwd = *bbp;
      *bbp = c_ptr;
      c_ptr->n_members = 0;
      c_ptr->bwd = NULL;
      strcpy(c_ptr->class_name, class_name);
      c_ptr->members = NULL;
    }
    else {
      printf("ERROR: Already MAXBB (%i) classes on the Blackboard!\n", MAXBB);
      return(-1);
    }
  }; /* Now we are sure that the class exists and c_ptr points to it */
  if ((m_ptr = find_mem(c_ptr->members, n_member->member_name)) == NULL) {
    /* new member */
    /**************/
    if (c_ptr->n_members < MAXMEM) {
/*      m_ptr = new_member(); */
      m_ptr = n_member;  /* use existing member: don't create a duplicate */
      c_ptr->n_members += 1;
      if (c_ptr->members != NULL) c_ptr->members->bwd = m_ptr;
/*      strcpy(m_ptr->member_name, n_member->member_name); */
      m_ptr->fwd = c_ptr->members;
      c_ptr->members = m_ptr;
      m_ptr->bwd = NULL;
/*      m_ptr->attributes = NULL;
      m_ptr->n_attr = 0; */
      /* the new member retains all its attributes ! */
    }
    else {
      printf("ERROR: Already MAXMEM (%i) members of class %s!\n",
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
      MAXMEM, c_ptr->class_name);
     return -1;
    }
  }
  else { /* member exists */
    rval = 1;
    /* Now insert values overwrite existing ones */
    for (a_ptr = n_member->attributes; a_ptr != NULL; a_ptr = a_ptr->fwd) {
      if ((a2_ptr = find_attr(m_ptr->attributes, a_ptr->attr_name)) == NULL) {
  /* new attribute */
  /*****************/
  if (m_ptr->n_attr < MAXATT) {
    a2_ptr = new_attr();
    m_ptr->n_attr += 1;
    if (m_ptr->attributes != NULL) m_ptr->attributes->bwd = a2_ptr;
    strcpy(a2_ptr->attr_name, a_ptr->attr_name);
    a2_ptr->fwd = m_ptr->attributes;
    m_ptr->attributes = a2_ptr;
    a2_ptr->bwd = NULL;
  }
  else {
    printf("ERROR: Already MAXATT (%i) attributes in member %s of class %s!\n",
          MAXATT, m_ptr->member_name, c_ptr->class_name);
    return -1;
  }
      };
      /* NOW COPY THE VALUE */
      strcpy(a2_ptr->attr_value.sval, a_ptr->attr_value.sval);
    }
    free(n_member); /* we copied everything */
  };
  return rval;
};

/* bb_delete removes a member from the bb. Returns 0 if successful,
   1 if member not found, 2 if class not found, -1 otherwise */
int bb_delete(bbp, class_name, member_name)
       bb_ptr bbp; char *class_name; char *member_name; {
  class_ptr c_ptr;
  member_ptr m_ptr;
  if ((c_ptr = bb_find_class(*bbp, class_name)) == NULL) return 2;
  if ((m_ptr = find_mem(c_ptr->members, member_name)) == NULL) return 1;
  if (m_ptr->bwd == NULL) {
    c_ptr->members = m_ptr->fwd;
  }
  else {
    m_ptr->bwd->fwd = m_ptr->fwd;
  };
  if (m_ptr->fwd != NULL) {
    m_ptr->fwd->bwd = m_ptr->bwd;
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
  );
  free(m_ptr);
  c_ptr->n_members -= 1;
);

/* THERE IS NO WAY TO REMOVE CLASSES (does not seem necessary) */
/* THERE IS NO WAY TO REMOVE ATTRIBUTES (does not seem necessary) */

/*********************************************************************/
/* Communication with knowledge sources */

/* add_attr creates a new attribute and adds it to a list of attributes.
   It returns 0 if successful, 1 if attr exists, -1 otherwise.  It does
   not test for overflow since it has no access to the member object.
   It sets the bwd pointer of the new element to NULL.  */
int add_attr(a_list, a_name, a_val)
      attr_ptr *a_list; char *a_name; char* a_val; {
  attr_ptr n_attr;
  if ((find_attr(*a_list, a_name)) != NULL) return 1;
  n_attr = new_attr();
  /* insert at start of list */
  n_attr->fwd = *a_list;
  if (*a_list != NULL) (*a_list)->bwd = n_attr;
  *a_list = n_attr;
  n_attr->bwd = NULL;
  strcpy(n_attr->attr_name, a_name);
  strcpy(n_attr->attr_value.sval, a_val); /* assume string value */
  return 0;
);

/* add_general_attr is a generalization of add_attr.  It allows any type of
   attribute value to be inserted. a_type indicates the type.  */
int add_general_attr(a_list, a_name, a_type, a_val)
      attr_ptr *a_list; char *a_name; int a_type; attr_val_type a_val; {
  attr_ptr n_attr;
  if ((find_attr(*a_list, a_name)) != NULL) return 1;
  n_attr = new_attr();
  /* insert at start of list */
  n_attr->fwd = *a_list;
  if (*a_list != NULL) (*a_list)->bwd = n_attr;
  *a_list = n_attr;
  n_attr->bwd = NULL;
  strcpy(n_attr->attr_name, a_name);
  switch (a_type) {
    case 1: n_attr->attr_value.ival = a_val.ival; break; /* integer */
    case 2: n_attr->attr_value.fval = a_val.fval; break; /* float */
    case 4: n_attr->attr_value.gptr = a_val.gptr; break; /* pointer */
    default: strcpy(n_attr->attr_value.sval, a_val.sval); /* assume string value */
      break;
    }
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
  return 0;
);
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
/* This file contains the subroutines required by the KES
   embedding methodology.
*/

#include "ssincl.h"

#ifdef MSDOS
char *position_cursor();
#endif

int load_kb(kb_name)
     char *kb_name;
{
   /* Load the parsed knowledge base */
   if (KES_ld_kb(strcat(kb_name,EXTENS), 50000L) != KES_success_c) {
     printf("Error loading knowledge base %s.");
     return(QUIT);
   }
   return(OK);
 }

/* When a KES function is called and it generates a message
   KES_receive_mesg() is called to display the message.  This function is a
   modified version of the function KES_receive_mesg() provided in the KES
   file psembed.c.  It prints all messages received except for those messages
   preceding a break in the knowledge base, because breaks are ignored in
   this program. */

void
KES_receive_mesg(message_text, message_class)
   KES_string_type message_text;        /* Actual message */
   KES_msg_class_type message_class;    /* Class of message */
{
   if ((strcmp(message_text, "\nType 'c' to begin\n") != 0) &&
       strcmp(message_text, "\nType 'n' for another case or 's' to stop\n") != 0 &&
       strncmp(message_text, "\nWarning", 8) != 0) {
#ifdef MSDOS
       display_as_message(message_text);
#endif
#ifdef HP-UX
       puts(message_text);      /* Print the message */
#endif
   }
}

/* KES calls KES_give_value_str() when it needs to determine the value for an
   attribute that either has no knowledge sources (an input attribute), or is
   being asked for explicitly.  The function below is a modified version of a
   sample function provided in the KES file psembed.c.  It accesses a
   simulated data base if the attribute cargo load is being asked for;
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
   otherwise the end user is asked for the value.  If the response is a why
   or explain, it executes it as a KES command; otherwise it checks if the
   response is a valid value for the attribute before returning it. */

KES_string_type
KES_give_value_str(attribute_desc)
   KES_atr_type    attribute_desc;      /* Points to an attribute */
   {
   /* Holds end user input.  Must be static because it is returned. */
   static char     response[LINE_LENGTH];
#ifdef MSDOS
   static char *prompt, str2[LINE_LENGTH];
#endif

   KES_string_type value;       /* For attribute value */

   /* Output parameter for KES_command() */
   KES_command_type command_type;

   /* Holds attribute value error message */
   KES_string_type error_mesg;

   for (;;) {                /* Loop until a valid attribute value is
                              input */

     /* Print attribute question prompt */
#ifdef HP-UX
     printf("%s", KES_q_askfor_prompt(attribute_desc));
     fflush(stdout);
#endif
#ifdef MSDOS
     prompt = KES_q_askfor_prompt(attribute_desc);
     display_as_dialogue(prompt);
     strcpy(str2, position_cursor(prompt));
     if (str2 == NULL) strcpy(str2, prompt);
     strcpy(str2, strcat(str2, " "));
#endif
     fgets(response, LINE_LENGTH, stdin);   /* Get end user response */

#ifdef MSDOS
     strcpy(str2, strcat(str2, response));
     display_as_dialogue(str2);
#endif

     response[strlen(response) - 1] = '\0'; /* Remove trailing newline */

     /* Execute why or explain command if entered.  strncmp() is used for
  explain since it may be followed by a number. */
     if (strcmp(response, "why") == 0 ||
   strncmp(response, "explain", 7) == 0) {
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
      /* The output parameter command_type is ignored here because we
    do not need to know what command was issued. */
#ifdef HP-UX
      puts(KES_command(response, &command_type));
#endif
#ifdef MSDOS
      display_as_message(KES_command(response, &command_type));
#endif
        }
     /* Check if response is a valid value for the attribute before
  returning it */
     else {
       error_mesg = KES_is_valid_value(attribute_desc, response);
       if (*error_mesg == '\0') {
   break;          /* A valid attribute value was given */
       } else {
#ifdef HP-UX
   puts(error_mesg);  /* Print error message */
#endif
#ifdef MSDOS
   display_as_message(error_mesg);
#endif
       }
     }
   }
   value = response; /* Assign end user input to attribute value */
   return (value);             /* Return attribute value */
 }

/* KES calls KES_q_members() when it needs to determine the members for a
   class that either has no knowledge sources, or is being asked for
   explicitly.  The function below is a modified version of a sample function
   provided in the KES file psembed.c.  It accesses a simulated data base if
   the members of the class Planes is asked for.  If the class being asked
   for is one other than Planes or Vehicles, the end user is asked for the
   members.  If the response is an explain, it executes it as a KES command;
   otherwise it checks if the response is a valid member list for the class
   before returning it. */

KES_string_type
KES_q_members(class_name)
   KES_string_type class_name;  /* Class name */
{
   /* Holds end user input.  Must be static because it is returned. */
   static char     response[LINE_LENGTH];
#ifdef MSDOS
   char *prompt, str2[LINE_LENGTH];
#endif
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
   KES_string_type member_names;/* For member list */

   /* Output parameter for KES_command() */
   KES_command_type command_type;

   /* Holds member list error message */
   KES_string_type error_mesg;

   for (;;) {                /* Loop until a valid member list is input */

     /* Print class question prompt */
#ifdef HP-UX
     printf("%s", KES_g_prompt_class(class_name));
     fflush(stdout);
#endif
#ifdef MSDOS
     prompt = KES_g_prompt_class(class_name);
     display_as_dialogue(prompt);
     strcpy(str2, position_cursor(prompt));
     if (str2 == NULL) strcpy(str2, prompt);
     strcpy(str2, strcat(str2, " "));
#endif
     fgets(response, LINE_LENGTH, stdin);   /* Get end user response */
#ifdef MSDOS
     strcpy(str2, strcat(str2, response));
     display_as_dialogue(str2);
#endif

     response[strlen(response) - 1] = '\0'; /* Remove trailing newline */

     /* Execute explain command if entered */
     if (strcmp(response, "explain") == 0) {

       /* The output parameter command_type is ignored here because we
    do not need to know what command was issued. */
#ifdef HP-UX
       puts(KES_command(response, &command_type));
#endif
#ifdef MSDOS
       display_as_message(KES_command(response, &command_type));
#endif
     }
     /* Check if response is a valid member list for the class before
  returning it */
     else {
       error_mesg = KES_is_valid_members(class_name, response,
                                         KES_false_c);
       if (*error_mesg == '\0') {
   break;          /* A valid member list was input */
       } else {
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
#ifdef HP-UX
   puts(error_mesg);/* Print error message */
#endif
#ifdef MSDOS
   display_as_message(error_mesg);
#endif
       }
     }
   }
   member_names = response;  /* Assign end user response to member names */
   return (member_names);       /* Return member names */
 }
#ifdef MSDOS
char *position_cursor(line)
  char *line; {
  char *last;
  int pos;
  last = strrchr(line, '\n');
  if (last == NULL) last = line;
  pos = strlen(last) + 5;
  _settextposition(11, pos);
  return last;
  }
#endif
```


ORIGINAL PAGE IS
OF POOR QUALITY

APPENDIX C: LISTING of EDIS KES CODE

ORIGINAL PAGE IS
OF POOR QUALITY

```
types:
  Size Type:  sql
 (A Little, Noticeably, A Lot).
  Direction Type: sql
        (Too High, Too Low, Garbage).
  Status Type: sql
 (Discovered, Verified, Ignored, Explained).
  S Change Type: sql
 (User, System).
  Inter Type: sql
 (yes, no).
  Relation Type: sql
 (proportional, inverse proportional).
%

attributes:
  Progress: truth [default: false].
  Find Primary Anomalies: truth [default: false].
  Internal Forward Propagation: truth [default: false].
  External Forward Propagation: truth [default: false].
  Energy coupling backward propagation: truth [default: false].
  diagnose: truth.
  Component With Top Anomaly: str.
  Parameter With Top Anomaly: str.
%
classes:
  COMPONENT: [default:CODummy]
    attributes:
      Name: str
 [default: ""]
 {explain: "The name of the component"}.
      ID: str
 {explain: "The identification of the actual component, e.g. serial number"}.
\ If ID = "" check the 'Is Part Of' Component!
\      Type: sql
\        (pump, turbine, pipe, valve, burner, sensor).
      State: sql
 (Assumed Good, Suspected, Known faulty, Known Good, Exonerated)
 [default: Assumed Good].
      Is Part Of: str
 [default: "none"]
 {explain: "Name of component this one is a part of"}.
      Is Composed Of: str
 [default: ""]
 {explain: "List of components of this component"}.
      Has Anomaly: truth
 [default: false].
      Has Top Anomaly: truth
 [default: true].
    %
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
    endclass.
  ABSTRACT_COMPONENT: [inherits: COMPONENT] [default: ACDummy]
\ a component which is not explicitly modeled except through its parts,
\ e.g. a turbopump.
    attributes:
      Function: str.
    %
    endclass.
  CONTROLLER: [inherits: COMPONENT] [default: CTDummy]
    attributes:
      Controlled Parameter: str.
      Actuated Parameter: str.
      Actuation Limit High: real.
      Actuation Limit Low: real.
      Relation: Relation Type
 {explain:
 "proportional: if actuation goes up, so does the controlled value"}.
      %
    endclass.
  THERMO_COMPONENT: [inherits: COMPONENT] [default: TDDummy]
    attributes:
      Medium: sql
 (LOX, Liquid Fuel, Partially Burned Fuel, Burned Gas).
      Medium Input: str
 [default: ""]
 {explain: "The name of the component attached to the input"}.
      Medium Output: str
 [default: ""]
 {explain: "The name of the component attached to the output"}.
      Medium Input Sensor: str
 [default: ""]
 {explain: "The name of a sensor at the input (or a list of names)"}.
      Medium Output Sensor: str
 [default: ""]
 {explain: "The name of a sensor at the output (or a list of names)"}.
      Internal Sensor: str
 [default: ""]
 {explain: "The name of an internal sensor"}.
      Has Input Anomaly: truth.
      Has Output Anomaly: truth.
    %
    endclass.
  SENSOR: [inherits: COMPONENT] [default: SEDummy]
    attributes:
      Component Name: str
 {explain: "The name of the nearest component"}.
      Location: sql
 (Inside, Medium_ Input, Medium_ Output, Energy Input, Energy Output)
 {explain: "Location relative to the component"}.
      Parameter Type: sql
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
(Temperature, Pressure, Flow Rate, Valve Position).
    Parameter Name: str
(explain: "The name of the measured parameter").
    Current Value: real.
    Current Value Is Anomalous: truth
[default: false].
    Sensor Type: sql
(Single, Redundant, Average)
(explain: "An 'average' sensor averages the readings",
         "from two or more 'redundant' sensors").
    Average Sensor Name: str
[default: ""]
(explain: "The name of the averaging sensor if this is a redundant sensor").
  %
  endclass.
TURBO_PUMP: [inherits: ABSTRACT_COMPONENT]
  attributes:
    Run Time: int
(explain: "Number of seconds it has run").
  %
  endclass.
MANIFOLD: [inherits: THERMO_COMPONENT]
  attributes:
    Number Of Inputs: int.
    Number Of Outputs: int.
\       (explain: "The names of the components attached to inputs and ouptuts",
\                 "are listed in 'Medium Input' and 'Medium Ouput' in the",
\                 "form of a character string separated by spaces.").
\    There is no way to attach sensors to a manifold in a sensible manner.
\    They have to be specified with the connected components.
  %
  endclass.
ENERGY_CONV_COMP: [inherits: THERMO_COMPONENT] [default: ECDummy]
  attributes:
    Efficiency: real.
    Power Coupled To: str
[default: ""]
(explain: "The name of the component which is coupled to this one").
    Power Direction: sql
(In, Out)
(explain: "in means energy is transferred to the medium,",
         "out the other way").
    Power Result: str
(explain: "The quantity which is affected by the power input").
\ e.g. for a pump this is the pressure difference!
    Coupling Sensor: str
[default: ""]
(explain: "A sensor which measures the energy coupling mechanism").
    Has Coupling Anomaly: truth.
  %
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
    endclass.
  PUMP: [inherits: ENERGY_CONV_COMP]
    endclass.
  TURBINE: [inherits: ENERGY_CONV_COMP] [default: TUDummy]
    endclass.
  GAS_TURBINE: [inherits: TURBINE]
    endclass.
  HYDRAULIC_TURBINE: [inherits: TURBINE]
    endclass.
  PIPE: [inherits: THERMO_COMPONENT]
    attributes:
      Normal Pressure Drop: real.
    %
    endclass.
  BURNER: [inherits: THERMO_COMPONENT]
    endclass.
  VALVE: [inherits: ENERGY_CONV_COMP]
\ Use SENSOR, TEST_DATA, COMPARISON_DATA and VARIATION LIMITS to
\ analyze valve performance. 'Power Coupled To' is the control input and
\ 'Coupling Sensor' is the position sensor.
    endclass.
  TEST_DATA:
    attributes:
      Parameter: str
 {explain: "Name of the parameter"}.
      Value: real
         {explain: "Value of the parameter"}.
      Interesting: sgl (yes, no)
         [default: no]
 {explain: "yes: if the value is useful for diagnosis"}.
    %
    endclass.
  COMPARISON_DATA:
    attributes:
      Parameter: str
 {explain: "Name of the parameter"}.
      Value: real
         {explain: "Value of the parameter"}.
    %
    endclass.
  ANOMALIES:
    attributes:
      Parameter: str.
      Size: Size Type.
      Direction: Direction Type.
      Status: Status Type.
      Status Change Time: int.
      Status Change Initiator: S Change Type
         [default: System].
      Explained By: str
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
 (explain: "The hypotheses which explain this anomaly"}.
      %
      endclass.
  HYPOTHESES:
    attributes:
      Explains Anomaly Of Parameter: str.
      Fault: sql
 (Unknown, Fluid Leak, Obstruction, Seal Leakage, Rotor Problem,
  Efficiency Problem).
      Faulty Component: str.
      Violated Behavior: str.
      %
    endclass.
%
rules:

  RB1:
    e:ENERGY_CONV_COMP
    if inclass(e, PUMP)
    then e>Power Direction = In.
    endif.
  RB2:
    e:ENERGY_CONV_COMP
    if inclass(e, GAS_TURBINE) or
       inclass(e, HYDRAULIC_TURBINE)
    then e>Power Direction = Out.
    endif.

  R1:
    c:COMPONENT
    if c>Has Top Anomaly = true
    then Component With Top Anomaly = c>Name.
         message combine ("The root anomaly appears to be at the ",
                          c>Name).
    endif.

\ If there is no anomaly directly associated with the component,
\ we cannot determine the parameter that is the root anomaly.
  R2:
    c:COMPONENT, s:SENSOR, a:ANOMALIES
    if Component With Top Anomaly = c>Name and
       s>Component Name = c>Name and
       a>Parameter = s>Parameter Name
    then Parameter With Top Anomaly = a>Parameter.
         message combine ("The parameter with the root anomaly is ",
                          a>Parameter).
    endif.
  R2a:
    c:CONTROLLER
    if Component With Top Anomaly = c>Name and
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
      inclass(c, CONTROLLER) = true
    then Parameter With Top Anomaly = c>Actuated Parameter.
         message combine ("The parameter with the root anomaly is ",
                          c>Actuated Parameter).
    endif.

\******
  R2b:
    c:THERMO_COMPONENT
    if c>Has Input Anomaly = true or
       c>Has Output Anomaly = true
    then c>Has Anomaly = true.
    endif.
  R2c:
    e:ENERGY_CONV_COMP
    if e>Has Coupling Anomaly
    then e>Has Anomaly = true.
    endif.

\*************************************************************************
\ Identify anomalous sensor readings
  RS1:
    s:SENSOR, a:ANOMALIES
    if s>Parameter Name = a>Parameter
    then s>Current Value Is Anomalous = true.
    endif.

\*************************************************************************
\ Primary anomalies:
\ Note the s>Current Value is Anomalous works but produces multiple
\ identical conclusions

  R3a1:
    c:THERMO_COMPONENT, s:SENSOR, a:ANOMALIES
    if Find Primary Anomalies = true and
       s>Component Name = c>Name and
\       s>Current Value Is Anomalous = true and
       s>Parameter Name = a>Parameter and
       s>Location = Medium_ Input
    then c>Has Input Anomaly = true.
         message combine (c>Name, " has input anomaly").
    endif.
  R3a2:
    c:THERMO_COMPONENT, s:SENSOR, a:ANOMALIES
    if Find Primary Anomalies = true and
       s>Component Name = c>Name and
\       s>Current Value Is Anomalous = true and
       s>Parameter Name = a>Parameter and
       s>Location = Medium_ Output
    then c>Has Output Anomaly = true.
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
        message combine (c>Name, " has output anomaly").
    endif.
  R3a3:
    e:ENERGY_CONV_COMP, s:SENSOR, a:ANOMALIES
    if Find Primary Anomalies = true and
       s>Component Name = e>Name and
\       s>Current Value Is Anomalous = true and
       s>Parameter Name = a>Parameter and
       s>Location = Energy Input or
       s>Location = Energy Output
    then e>Has Coupling Anomaly = true.
        message combine (e>Name, " has coupling anomaly").
    endif.

  R3a4:
    c:CONTROLLER, a:ANOMALIES
    if Find Primary Anomalies = true and
       c>Actuated Parameter = a>Parameter or
       c>Controlled Parameter = a>Parameter
    then c>Has Anomaly = true.
        message combine ("Controller ", c>Name,
                         " is involved with anomalous values").
    endif.
\**********
\ Identify correct parameters: sensors do not imply an anomaly

  R3a5:
    c:THERMO_COMPONENT, s:SENSOR, a:ANOMALIES
    if Find Primary Anomalies = true and
       s>Component Name = c>Name and
       s>Current Value Is Anomalous = false and
\       s>Parameter Name = a>Parameter and
       s>Location = Medium_ Input
    then c>Has Input Anomaly = false.
\        message combine (c>Name, " input ok").
    endif.
  R3a6:
    c:THERMO_COMPONENT, s:SENSOR, a:ANOMALIES
    if Find Primary Anomalies = true and
       s>Component Name = c>Name and
       s>Current Value Is Anomalous = false and
\       s>Parameter Name = a>Parameter and
       s>Location = Medium_ Output
    then c>Has Output Anomaly = false.
\        message combine (c>Name, " output ok").
    endif.
  R3a7:
    e:ENERGY_CONV_COMP, s:SENSOR, a:ANOMALIES
    if Find Primary Anomalies = true and
       s>Component Name = e>Name and
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
      s>Current Value Is Anomalous = false and
\      s>Parameter Name = a>Parameter and
      s>Location = Energy Input or
      s>Location = Energy Output
   then e>Has Coupling Anomaly = false.
\        message combine (e>Name, " coupling ok").
   endif.


\***********
\ Internal forward propagation: assume if in is anomalous, out is too
\ unless otherwise known.  This will include too many components,
\ but that is no problem.

  R3b1:
    t:THERMO_COMPONENT
    if Internal Forward Propagation = true and
       t>Has Input Anomaly = true
    then reassert t>Has Output Anomaly = true <0.9>.
         message combine (t>Name, " has output anomaly").
    endif.
  R3b2:
    e:ENERGY_CONV_COMP
    if Internal Forward Propagation = true and
       e>Has Coupling Anomaly = true
    then reassert e>Has Output Anomaly = true <0.9>.
         message combine (e>Name, " has output anomaly").
    endif.
  R3b3:
    e:ENERGY_CONV_COMP
    if Internal Forward Propagation = true and
       e>Has Input Anomaly = true and
       e>Power Direction = Out
    then reassert e>Has Coupling Anomaly = true <0.9>.
         message combine (e>Name, " has coupling anomaly").
    endif.

\***********
\External forward propagation

  R3c1:
    t:THERMO_COMPONENT, t2:THERMO_COMPONENT
    if External Forward Propagation = true and
       t>Medium Input = t2>Name and
       t2>Has Output Anomaly = true
    then t>Has Input Anomaly = true <0.9>.
         message combine (t>Name, " has input anomaly").
    endif.
  R3c2:
    e:ENERGY_CONV_COMP, e2:ENERGY_CONV_COMP
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
    if External Forward Propagation = true and
       e>Power Direction = In and
       e>Power Coupled To = e2>Name and
       e2>Has Coupling Anomaly = true
    then e>Has Coupling Anomaly = true <0.9>.
         message combine (e>Name, " has coupling anomaly").
    endif.

\*********
\ Energy coupling backward propagation

  R3d1:
    e:ENERGY_CONV_COMP, e2:ENERGY_CONV_COMP
    if Energy coupling backward propagation = true and
       e>Power Direction = Out and
       e>Power Coupled To = e2>Name and
       e2>Has Coupling Anomaly = true
    then e>Has Coupling Anomaly = true <0.9>.
         message combine (e>Name, " has coupling anomaly").
    endif.



\*************************************************************************
\ Find top anomalies (most upstream fault manifestations)

  R4:
    c:COMPONENT
    if c>Has Anomaly = false
    then c>Has Top Anomaly = false.
    endif.

  R5:
    t:THERMO_COMPONENT, c:COMPONENT
    if t>Has Anomaly = true and
       t>Medium Input # "" and
       t>Medium Input = c>Name and
       c>Has Anomaly = true
    then t>Has Top Anomaly = false.
         message combine ("Anomalies of ", t>Name,
                          " may be caused by ", c>Name).
    endif.

  R6:
    e:ENERGY_CONV_COMP, c:COMPONENT
    if e>Has Anomaly = true and
       e>Power Direction = In and
       e>Power Coupled To = c>Name and
       c>Has Anomaly = true
    then e>Has Top Anomaly = false.
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
        message combine ("Anomalies of ", e>Name,
                         " may be caused by ", c>Name).
    endif.

  R7a:
    c:CONTROLLER, a:ANOMALIES
    if c>Has Anomaly = true and
       c>Controlled Parameter = a>Parameter
    then c>Has Top Anomaly = false.
        message combine ("Anomalies of ", c>Name,
                         " may be caused by ", c>Controlled Parameter).
    endif.
\ If the controller adjusts the controlled parameter to the correct
\ value, it is not considered a root anomaly although it is the most
\ upstream component in the physical chain.
  R7b:
    c:CONTROLLER, s:SENSOR
    if c>Has Anomaly = true and
       s>Parameter Name = c>Controlled Parameter and
       s>Current Value Is Anomalous = false
    then c>Has Top Anomaly = false.
        message combine ("Anomalies of ", c>Name,
                         " may be caused by ", c>Controlled Parameter).
    endif.

\  R8: not necessary if only valves are controlled: see VALVE
\    c:CONTROLLER, c1:COMPONENT, s:SENSOR
\    if c1>Has Anomaly = true and
\       s>Component Name = c1>Name and
\       s>Parameter Name = c>Actuated Parameter and
\       c>Actuated Parameter = a>Parameter
\    then c1>Has Top Anomaly = false.

%

demons:

  act demon:
    when
      diagnose = true
    then
      message
 combine ("Beginning Diagnosis").
\unfortunately: we have to take care of the dummy members
       COMPONENT:CODummy>Has Anomaly = false.
 ABSTRACT_COMPONENT:ACDummy>Has Anomaly = false.
\       CONTROLLER:CTDummy>Has Anomaly = false.
 THERMO_COMPONENT:TDDummy>Has Anomaly = false.
\       SENSOR:SEDummy>Has Anomaly = false.
 ENERGY_CONV_COMP:ECDummy>Has Anomaly = false.
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
TURBINE:TUDummy>Has Anomaly = false.
\       message "********  Classifying Sensor Readings  ********".
       forall s:SENSOR do
         obtain s>Current Value Is Anomalous.
       endforall.
\  break.
       message "********  Finding primary anomalies  ********".
       Find Primary Anomalies = true.
        forall c:THERMO_COMPONENT do
          obtain c>Has Input Anomaly.
          obtain c>Has Output Anomaly.
        endforall.
        forall c:ENERGY_CONV_COMP do
          obtain c>Has Coupling Anomaly.
        endforall.
        forall c:COMPONENT do
          obtain c>Has Anomaly.
        endforall.
       reassert Find Primary Anomalies = false.
       Progress = true.
     while Progress = true do
       reassert Progress = false.
       message "********  Internal forward propagation  ********".
       reassert Internal Forward Propagation = true.
       forall c:THERMO_COMPONENT do
         if status(c>Has Output Anomaly) = unknown then
           erase c>Has Anomaly.
           erase c>Has Output Anomaly.
\          obtain c>Has Output Anomaly.
           obtain c>Has Anomaly.
           if status(c>Has Output Anomaly) = known then
             reassert Progress = true.
           endif.
         endif.
       endforall.
       forall c:ENERGY_CONV_COMP do
         if status(c>Has Coupling Anomaly) = unknown then
           erase c>Has Anomaly.
           erase c>Has Coupling Anomaly.
\          obtain c>Has Coupling Anomaly.
           obtain c>Has Anomaly.
           if status(c>Has Coupling Anomaly) = known then
             reassert Progress = true.
           endif.
         endif.
       endforall.
       reassert Internal Forward Propagation = false.
       message "********  External forward propagation  ********".
       reassert External Forward Propagation = true.
       forall c:THERMO_COMPONENT do
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
        if status(c>Has Input Anomaly) = unknown then
          erase c>Has Anomaly.
          erase c>Has Input Anomaly.
          obtain c>Has Input Anomaly.
          obtain c>Has Anomaly.
          if status(c>Has Input Anomaly) = known then
            reassert Progress = true.
          endif.
        endif.
      endforall.
      forall c:ENERGY_CONV_COMP do
        if status(c>Has Coupling Anomaly) = unknown and
           c>Power Direction = In then
          erase c>Has Anomaly.
          erase c>Has Coupling Anomaly.
          obtain c>Has Coupling Anomaly.
          obtain c>Has Anomaly.
          if status(c>Has Coupling Anomaly) = known then
            reassert Progress = true.
          endif.
        endif.
      endforall.
    endwhile.
      reassert External Forward Propagation = false.
      message "********  Energy coupling backward propagation  ********".
      reassert Energy coupling backward propagation = true.
      forall e:ENERGY_CONV_COMP do
        if status(e>Has Coupling Anomaly) = unknown and
           e>Power Direction = Out then
          erase e>Has Anomaly.
          erase e>Has Coupling Anomaly.
          obtain e>Has Anomaly.
        endif.
      endforall.
      reassert Energy coupling backward propagation = false.

      message "The following components show or are expected to show".
      message "anomalous values:".
      obtain Parameter With Top Anomaly.

      if status(Component With Top Anomaly) = unknown then
           we were unsuccessful!
        forall c:CONTROLLER do
          if c>Has Anomaly = true then
            message combine ("The anomaly is situated between ",
                             c>Actuated Parameter, " and ",
                             c>Controlled Parameter).
            reassert Parameter With Top Anomaly = c>Actuated Parameter.
          endif.
        endforall.
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
        endif.
    endwhen.

%
actions:
   message "These actions will not be used".

assertclass PUMP = LPFP, HPFP, LPOP, HPOP.
PUMP:LPFP>Name = "LPFP".
PUMP:LPFP>Is Part Of = "LPFTP".
PUMP:LPFP>Medium = Liquid Fuel.
PUMP:LPFP>Medium Input = "".
PUMP:LPFP>Medium Output = "F101".
PUMP:LPFP>Medium Input Sensor = "LPFP_FUEL_IN_PR_S LPFP_FUEL_IN_TEMP_S".
PUMP:LPFP>Medium Output Sensor = "".
PUMP:LPFP>Internal Sensor = "".
PUMP:LPFP>Efficiency = 1.0.
PUMP:LPFP>Power Coupled To = "LPFT".
PUMP:LPFP>Power Direction = In.
PUMP:LPFP>Coupling Sensor = "S2".

PUMP:HPFP>Name = "HPFP".
PUMP:HPFP>Medium = Liquid Fuel.
PUMP:HPFP>Medium Input = "F101".
PUMP:HPFP>Medium Output = "F102".
PUMP:HPFP>Medium Input Sensor = "".
\       "HPFP_IN_PRESS_S HPFP_IN_TEMP_S HPFP_FUEL_FLOW_S".
PUMP:HPFP>Medium Output Sensor = "HPFP_DISCH_PR_S HPFP_DISCH_TEMP_S".
PUMP:HPFP>Internal Sensor = "".
PUMP:HPFP>Efficiency = 1.0.
PUMP:HPFP>Power Coupled To = "HPFT".
PUMP:HPFP>Power Direction = In.
PUMP:HPFP>Coupling Sensor = "HPFT_SHAFT_SPEED_S".

PUMP:LPOP>Name = "LPOP".
PUMP:LPOP>Medium = LOX.
PUMP:LPOP>Medium Input = "".
PUMP:LPOP>Medium Output = "O301".
PUMP:LPOP>Power Coupled To = "LPOT".
PUMP:LPOP>Power Direction = In.

PUMP:HPOP>Name = "LPOP".
PUMP:HPOP>Medium = LOX.

assertclass HYDRAULIC_TURBINE = LPFT, LPOT.
HYDRAULIC_TURBINE:LPFT>Name = "LPFT".
HYDRAULIC_TURBINE:LPFT>Power Coupled To = "LPFP".
HYDRAULIC_TURBINE:LPFT>Coupling Sensor = "".
HYDRAULIC_TURBINE:LPFT>Medium = Liquid Fuel.
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
HYDRAULIC_TURBINE:LPOT>Name = "LPOT".
HYDRAULIC_TURBINE:LPOT>Power Coupled To = "LPOP".
HYDRAULIC_TURBINE:LPOT>Coupling Sensor = "".
HYDRAULIC_TURBINE:LPOT>Medium = LOX.

assertclass GAS_TURBINE = HPFT, HPOT.
GAS_TURBINE:HPFT>Name = "HPFT".
GAS_TURBINE:HPFT>Medium Input = "FPB".
GAS_TURBINE:HPFT>Power Direction = Out.
GAS_TURBINE:HPFT>Power Coupled To = "HPFP".

GAS_TURBINE:HPOT>Name = "HPOT".

assertclass TURBO_PUMP = LPFTP, HPFTP, LPOTP, HPOTP.

TURBO_PUMP:LPFTP>Name = "LPFTP".
TURBO_PUMP:LPFTP>Is Composed Of = "LPFP LPFT".

TURBO_PUMP:HPFTP>Name = "HPFTP".
TURBO_PUMP:HPFTP>Is Composed Of = "HPFP HPFT".

TURBO_PUMP:LPOTP>Name = "LPOTP".
TURBO_PUMP:LPOTP>Is Composed Of = "LPOP LPOT".

TURBO_PUMP:HPOTP>Name = "HPOTP".
TURBO_PUMP:HPOTP>Is Composed Of = "HPOP HPOT".

TURBO_PUMP:LPFTP>ID = "2411R1".
TURBO_PUMP:HPFTP>ID = "4306".
TURBO_PUMP:LPOTP>ID = "2311".
TURBO_PUMP:HPOTP>ID = "0710".

assertclass VALVE = OPOV, FPOV.
VALVE:OPOV>Name = "OPOV".
VALVE:FPOV>Name = "FPOV".
VALVE:FPOV>Power Direction = In.
VALVE:FPOV>Power Coupled To = "EC1".
VALVE:FPOV>Medium Output = "FPB".

assertclass MANIFOLD = MAN1.
MANIFOLD:MAN1>Name = "MAN1".

assertclass PIPE = F101, O101.
PIPE:F101>Name = "F101".
PIPE:F101>Medium Input = "LPFP".
PIPE:F101>Medium Output = "HPFP".

PIPE:O101>Name = "O101".

assertclass BURNER = FPB, OPB.
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
BURNER:FPB>Name = "FPB".
BURNER:FPB>Medium Input = "FPOV".
BURNER:FPB>Medium = Partially Burned Fuel.
BURNER:FPB>Medium Output = "HPFT".

BURNER:OPB>Name = "OPB".

assertclass SENSOR = S1, S2, S3, S4, S5, S6.
SENSOR:S1>Name = "S1".
SENSOR:S1>Component Name = "LPFP".
SENSOR:S1>Parameter Name = "LPFP_FUEL_DISCH_PR".
SENSOR:S1>Location = Medium_ Output.
SENSOR:S2>Name = "S2".
SENSOR:S2>Component Name = "LPFP".
SENSOR:S2>Parameter Name = "LPFP_SHAFT_SPEED".
SENSOR:S2>Location = Energy Input.
SENSOR:S3>Name = "S3".
SENSOR:S3>Component Name = "FPOV".
SENSOR:S3>Parameter Name = "FPOV_POSITION".
SENSOR:S3>Location = Energy Input.
SENSOR:S4>Name = "S4".
SENSOR:S4>Component Name = "HPFT".
SENSOR:S4>Parameter Name = "HPFT_DISCH_TEMP".
SENSOR:S4>Location = Medium_ Output.
SENSOR:S5>Name = "S5".
SENSOR:S5>Component Name = "HPFP".
SENSOR:S5>Parameter Name = "HPFP_DISCH_PR".
SENSOR:S5>Location = Medium_ Output.
SENSOR:S6>Name = "S6".
SENSOR:S6>Component Name = "HPFP".
SENSOR:S6>Parameter Name = "HPFP_IN_PR".
SENSOR:S6>Location = Medium_ Input.

assertclass CONTROLLER = EC1.
CONTROLLER:EC1>Name = "EC1".
CONTROLLER:EC1>Controlled Parameter = "HPFP_DISCH_PR".
CONTROLLER:EC1>Actuated Parameter = "FPOV_POSITION".

assertclass ANOMALIES = a1, a2, a3, a4, a5.
ANOMALIES:a1>Parameter = "LPFP_FUEL_DISCH_PR".
ANOMALIES:a2>Parameter = "HPFT_DISCH_TEMP".
ANOMALIES:a3>Parameter = "FPOV_POSITION".
ANOMALIES:a4>Parameter = "LPFP_SHAFT_SPEED".
ANOMALIES:a5>Parameter = "HPFP_IN_PR".
diagnose = true.
}
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
types:

  Size Type: sql
 (A Little, Noticeably, A Lot).

  Direction Type: sql
       (Too High, Too Low, Garbage).

  Status Type: sql
 (Discovered, Verified, Ignored, Explained).

  S Change Type: sql
 (User, System).

  Inter Type: sql
 (yes, no).
%
attributes:

  find anomalies: truth.

  Finished: truth.

  T Size: Size Type.

  T Direction: Direction Type.

  T Status: Status Type.

  T Value: real.

  C Value: real.

  V Normal Variation: real.

  V Small Anomaly: real.

  V Medium Anomaly: real.

  T Interesting: Inter Type.

  Difference: real
 [default: (T Value - C Value)].

Counter: int.

%
classes:

  TEST_DATA:
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
attributes:

     Parameter: str
 (explain: "Name of the parameter").

     Value: real
        (explain: "Value of the parameter").

     Interesting: Inter Type
        [default: no]
 (explain: "yes: if the value is useful for diagnosis").
%

   endclass.

  COMPARISON_DATA:
attributes:

     Parameter: str
 (explain: "Name of the parameter").

     Value: real
        (explain: "Value of the parameter").
%

   endclass.

  VARIATION_LIMITS:
attributes:

     Parameter: str.

     Sensor: str
 (explain: "The sensor which measures the parameter").

     Normal Variation: real
 (explain: "Absolute value variation which is still considered normal").

     Small Anomaly: real
 (explain: "Limit for a deviation considered small").

     Medium Anomaly: real
 (explain: "A larger deviation will be considered large").
%

   endclass.

  ANOMALIES:
attributes:
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
      Parameter: str.

\        [default: ""].
      Size: Size Type.

      Direction: Direction Type.

      Status: Status Type.

      Status Change Time: int.

      Status Change Initiator: S Change Type
         [default: System].

      Explained By: str
 (explain: "The hypotheses which explain this anomaly").
%

      endclass.
%
rules:

  Size Rule1:
    if
      V Normal Variation lt  abs(Difference) and
      V Small Anomaly ge abs(Difference)
    then
      T Size = A Little.
      message "Size = A Little".
     endif.

  Size Rule2:
    if
      V Small Anomaly lt  abs(Difference) and
      V Medium Anomaly ge abs(Difference)
    then
      T Size = Noticeably.
      message "Size = Noticeably".
     endif.

  Size Rule3:
    if
      V Medium Anomaly lt  abs(Difference)
    then
      T Size = A Lot.
      message "Size = A Lot".
     endif.

  Direction Rule High:
    if
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
      Difference gt (0) and
        T Size = A Little or
        T Size = Noticeably or
        T Size = A Lot
    then
      message "Dir = Too High".
      T Direction = Too High.
    endif.

  Direction Rule Low:
    if
      Difference lt (0) and
        T Size = A Little or
        T Size = Noticeably or
        T Size = A Lot
    then
      message "Dir = Too Low".
      T Direction = Too Low.
    endif.

  Interesting Data Rule:
    if
       T Direction = Too High or
       T Direction = Too Low
    then
      T Interesting = yes.
    endif.
%
demons:

  Doit:
    when
      find anomalies = true
    then
      Counter = 0.
      forall t:TEST_DATA do
        forall c:COMPARISON_DATA do
          if t>Parameter = c>Parameter then
            forall v:VARIATION_LIMITS do
              if t>Parameter = v>Parameter then
                T Value = t>Value.
                C Value = c>Value.
                V Normal Variation = v>Normal Variation.
                V Small Anomaly = v>Small Anomaly.
                V Medium Anomaly = v>Medium Anomaly.
                if T Interesting = yes then
                   reassert Counter = (Counter + 1).
                   addmember ANOMALIES, combine("anomaly", Counter).
                  reassert Finished = false.
                  forall a:ANOMALIES do
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
                    if determined(a>Parameter) = false and
                       Finished = false
                    then
                      message combine ("Storing in ", a).
                      a>Parameter = t>Parameter.
                      a>Size = T Size.
                      a>Direction = T Direction.
                      a>Status = Discovered.
                      a>Status Change Initiator = System.
                      reassert Finished = true.
                    endif.
                  endforall.
                endif.
                erase T Value, C Value, V Medium Anomaly, T Interesting, T Size.
                erase T Direction, V Normal Variation, V Small Anomaly, Difference.
              endif.
            endforall.
          endif.
        endforall.
      endforall.
    endwhen.

%
actions:

  message "These actions will not be used".

      read "dvarlm.dat",
 VARIATION_LIMITS, VARIATION_LIMITS (Parameter, Sensor,
        Normal Variation, Small Anomaly, Medium Anomaly).
\ message combine ("VLimit: LPFP_FUEL_IN_PR n var: ",
\       VARIATION_LIMITS:LPFP_FUEL_IN_PR>Normal Variation).
      message
 combine ("Reading test data file: tdata.dat").
\ the data files may later be replaced by raw data files and read
\       by a C function!
      read "tdata.dat",
 TEST_DATA, TEST_DATA (Parameter, Value).
\ message combine ("TData: Value: ", TEST_DATA:LPFP_FUEL_IN_PR>Value).
      message
 combine ("Reading comparison data file: cdata.dat").
      read "cdata.dat",
 COMPARISON_DATA, COMPARISON_DATA (Parameter, Value).

find anomalies = true.
%
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
types:

  Relation Type: sql
 (proportional, inverse proportional).
%
attributes:

  reading files: truth.

  GCFile: str.

  SCFile: str.

  VLFile: str.

  TDFile: str.

  CDFile: str.
%
classes:

  COMPONENT:
attributes:

     Name: str
 (explain: "The name of the component").

     ID: str
 (explain: "The identification of the actual component, e.g. serial number").

\ If ID = "" check the 'Is Part Of' Component!
\     Type: sql
\       (pump, turbine, pipe, valve, burner, sensor).
     State: sql
 (Assumed Good, Suspected, Known faulty, Known Good, Exonerated)
 [default: Assumed Good].

     Is Part Of: str
 [default: "none"]
 (explain: "Name of component this one is a part of").

     Is Composed Of: str
 [default: ""]
 (explain: "List of components of this component").
%

   endclass.

  ABSTRACT_COMPONENT: [inherits: COMPONENT]
attributes:
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
     Function: str.
%

    endclass.


  CONTROLLER: [inherits: COMPONENT]
attributes:

     Controlled Parameter: str.

     Actuated Parameter: str.

     Actuation Limit High: real.

     Actuation Limit Low: real.

     Relation: Relation Type
 {explain:
 "proportional: if actuation goes up, so does the controlled value"}.
%

    endclass.

  THERMO_COMPONENT: [inherits: COMPONENT]
attributes:

     Medium: sql
 (LOX, Liquid Fuel, Partially Burned Fuel, Burned Gas).

     Medium Input: str
 {explain: "The name of the component attached to the input"}.

     Medium Output: str
 {explain: "The name of the component attached to the output"}.

     Medium Input Sensor: str
 {explain: "The name of a sensor at the input (or a list of names)"}.

     Medium Output Sensor: str
 {explain: "The name of a sensor at the output (or a list of names)"}.

     Internal Sensor: str
 {explain: "The name of an internal sensor"}.
%

    endclass.

   SENSOR: [inherits: COMPONENT]
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
attributes:

     Component Name: str
 (explain: "The name of the nearest component").

     Location: sql
 (Inside, Medium_ Input, Medium_ Output, Energy Input, Energy Output)
 (explain: "Location relative to the component").

     Parameter Type: sql
 (Temperature, Pressure, Flow Rate, Valve Position).

     Parameter Name: str
 (explain: "The name of the measured parameter").

     Current Value: real.

     Sensor Type: sql
 (Single, Redundant, Average)
 (explain: "An 'average' sensor averages the readings",
          "from two or more 'redundant' sensors").

     Average Sensor Name: str
 (explain: "The name of the averaging sensor if this is a redundant sensor").
}

   endclass.

  TURBO_PUMP: [inherits: ABSTRACT_COMPONENT]
attributes:

     Run Time: int
 (explain: "Number of seconds it has run").
}

   endclass.

  MANIFOLD: [inherits: THERMO_COMPONENT]
attributes:

     Number Of Inputs: int.

     Number Of Outputs: int.
}

   endclass.

  ENERGY_CONV_COMP: [inherits: THERMO_COMPONENT]
attributes:
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
     Efficiency: real.

     Power Coupled To: str
 (explain: "The name of the component which is coupled to this one").

     Power Direction: sql
 (In, Out)
 (explain: "In means energy is transferred to the medium,",
           "Out the other way").

     Power Result: str
 (explain: "The quantity which is affected by the power input").

\ e.g. for a pump this is the pressure difference!
     Coupling Sensor: str
 (explain: "A sensor which measures the energy coupling mechanism").
%

   endclass.

  PUMP: [inherits: ENERGY_CONV_COMP]

   endclass.

  TURBINE: [inherits: ENERGY_CONV_COMP]

   endclass.

  GAS_TURBINE: [inherits: TURBINE]

   endclass.

  HYDRAULIC_TURBINE: [inherits: TURBINE]

   endclass.

  PIPE: [inherits: THERMO_COMPONENT]
attributes:

     Normal Pressure Drop: real.
%

   endclass.

  BURNER: [inherits: THERMO_COMPONENT]

   endclass.

  VALVE: [inherits: ENERGY_CONV_COMP]
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
    endclass.

  FILE:
attributes:

     Name: str.

     Type: sql
 (general configuration, specific configuration, variation limits,
 test data, comparison data).

     Comparison Type: sql
 (none, previous test, average data, two sigma limit, absolute limit).
%

   endclass.

\ file names are read from BB
  TEST_DATA:
attributes:

     Parameter: str
 (explain: "Name of the parameter").

     Value: real
        (explain: "Value of the parameter").

     Interesting: sql (yes, no)
        [default: no]
 (explain: "yes: if the value is useful for diagnosis").
%

   endclass.

  COMPARISON_DATA:
attributes:

     Parameter: str
 (explain: "Name of the parameter").

     Value: real
        (explain: "Value of the parameter").
%

   endclass.

  VARIATION_LIMITS:
attributes:

     Parameter: str.
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
     Sensor: str
 (explain: "The sensor which measures the parameter").

     Normal Variation: real
 (explain: "Absolute value variation which is still considered normal").

     Small Anomaly: real
 (explain: "Limit for a deviation considered small").

     Medium Anomaly: real
 (explain: "A larger deviation will be considered large").
%

   endclass.
%
rules:

  find general configuration File:
    f:FILE
    if
      f > Type = general configuration
    then
      GCFile = f>Name.
    endif.

  find specific configuration File:
    f:FILE
    if
      f > Type = specific configuration
    then
      SCFile = f>Name.
    endif.

  find var lim File:
    f:FILE
    if
      f > Type = variation limits
    then
      VLFile = f>Name.
    endif.

  find test data File:
    f:FILE
    if
      f > Type = test data
    then
      TDFile = f>Name.
    endif.
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
  find comparison data File:
    f:FILE
    if
      f > Type = comparison data
    then
      CDFile = f>Name.
    endif.
}
demons:

  act demon:
    when
      reading files = true
    then
      message
 combine ("Reading general configuration file: ", GCFile).
      read GCFile,
           PUMP, PUMP (Name, Is Part Of, Is Composed Of,
                       Medium, Medium Input, Medium Output,
                  Medium Input Sensor, Medium Output Sensor,
                  Internal Sensor, Efficiency, Power Coupled To,
                  Power Direction, Coupling Sensor),
    GAS_TURBINE, GAS_TURBINE (Name, Is Part Of, Is Composed Of,
                       Medium, Medium Input, Medium Output,
                  Medium Input Sensor, Medium Output Sensor,
                  Internal Sensor, Efficiency, Power Coupled To,
                  Power Direction, Coupling Sensor),
    HYDRAULIC_TURBINE,
    HYDRAULIC_TURBINE (Name, Is Part Of, Is Composed Of,
                  Medium, Medium Input, Medium Output,
                  Medium Input Sensor, Medium Output Sensor,
                  Internal Sensor, Efficiency, Power Coupled To,
                  Power Direction, Coupling Sensor),
    TURBO_PUMP, TURBO_PUMP (Name, Is Part Of, Is Composed Of, Function),
    PIPE, PIPE (Name, Is Part Of, Is Composed Of,
                       Medium, Medium Input, Medium Output,
                  Medium Input Sensor, Medium Output Sensor,
                  Internal Sensor, Normal Pressure Drop),
    MANIFOLD, MANIFOLD (Name, Is Part Of, Is Composed Of,
                       Medium, Medium Input, Medium Output,
                  Medium Input Sensor, Medium Output Sensor,
                  Internal Sensor, Number Of Inputs, Number Of Outputs),
    SENSOR, SENSOR (Name, Is Part Of, Is Composed Of,
                       Component Name, Location, Parameter Type,
                       Parameter Name, Sensor Type, Average Sensor Name),
           BURNER, BURNER (Name, Is Part Of, Is Composed Of,
                       Medium, Medium Input, Medium Output,
                  Medium Input Sensor, Medium Output Sensor,
                  Internal Sensor),
           VALVE, VALVE (Name, Is Part Of, Is Composed Of,
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
                    Medium, Medium Input, Medium Output,
                Medium Input Sensor, Medium Output Sensor,
                Internal Sensor, Efficiency, Power Coupled To,
                Power Direction, Coupling Sensor),
    CONTROLLER, CONTROLLER (Name, Controlled Parameter,
                Actuated Parameter).
\ message combine ("GConf: PUMP Name: ", PUMP:LPFP>Name).
      message
 combine ("Reading specific configuration file: ", SCFile).
      read SCFile,
 TURBO_PUMP (ID, Run Time).
\ message combine ("SConf: TURBO_PUMP ID: ", TURBO_PUMP:LPFTP>ID).
      message
 combine ("Reading variation limit file: ", VLFile).
      read VLFile,
 VARIATION_LIMITS, VARIATION_LIMITS (Parameter, Sensor,
        Normal Variation, Small Anomaly, Medium Anomaly).
\ message combine ("VLimit: LPFP_FUEL_IN_PR n var: ",
\       VARIATION_LIMITS:LPFP_FUEL_IN_PR>Normal Variation).
      message
 combine ("Reading test data file: ", TDFile).
\ the data files may later be replaced by raw data files and read
\       by a C function!
      read TDFile,
 TEST_DATA, TEST_DATA (Parameter, Value).
\ message combine ("TData: Value: ", TEST_DATA:LPFP_FUEL_IN_PR>Value).
      message
 combine ("Reading comparison data file: ", CDFile).
      read CDFile,
 COMPARISON_DATA, COMPARISON_DATA (Parameter, Value).
\ message combine ("CData: Value: ", COMPARISON_DATA:LPFP_FUEL_IN_PR>Value).
    break.
    endwhen.

%
actions:

   message "These actions will not be used".
   addmember FILE, "file1", "file2", "file3", "file4", "file5".
   FILE:file1>Type = general configuration.
   FILE:file1>Name = "gconf.dat".
   FILE:file1>Comparison Type = none.

   FILE:file2>Name = "sconf.dat".
   FILE:file2>Type = specific configuration.
   FILE:file2>Comparison Type = none.

   FILE:file3>Type = variation limits.
   FILE:file3>Name = "dvarlm.dat".
   FILE:file3>Comparison Type = none.
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
   FILE:file4>Type = test data.
   FILE:file4>Name = "tdata.dat".
   FILE:file4>Comparison Type = none.

   FILE:file5>Type = comparison data.
   FILE:file5>Name = "cdata.dat".
   FILE:file5>Comparison Type = previous test.

reading files = true.

}
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
\ KB to create strategy

constants:

\ The following constants are used for messages in the actions section.

   welcome:
      " ",
      "Welcome to EDIS.",
      " ".
   banner:
      "**************************************************************************".
%

attributes:

   find task: sgl (done, cannot find).
%

classes:

   TASK:
      attributes:
 time: int.
 priority: int.
 task name: str.
 knowledge source: str.

 %
      endclass.

   REQUEST:
      attributes:
 bogus: int.
 %
      endclass.

   OFFER:
      attributes:
 bogus: int.
 %
      endclass.

%

rules:

   Create tasks:
      if
 true
```


ORIGINAL PAGE IS
OF POOR QUALITY

```
     then
find task = done.
     endif.

%
demons:
  D1:
    when
find task = done
    then
addmember TASK, "t1", "t2", "t3", "t4", "t5", "t6".
TASK:t1 > time = 1.
TASK:t1 > priority = 50.
TASK:t1 > task name = "greet user".
TASK:t1 > knowledge source = "user_IF1".
TASK:t4 > time = 1.
TASK:t4 > priority = 47.
TASK:t4 > task name = "get file name".
TASK:t4 > knowledge source = "user_IF2".
TASK:t3 > time = 1.
TASK:t3 > priority = 45.
TASK:t3 > task name = "read data files".
TASK:t3 > knowledge source = "file reader".
TASK:t5 > time = 1.
TASK:t5 > priority = 40.
TASK:t5 > task name = "find anomalies".
TASK:t5 > knowledge source = "data analyzer".
TASK:t6 > time = 1.
TASK:t6 > priority = 35.
TASK:t6 > task name = "diagnose".
TASK:t6 > knowledge source = "diagnostician".
TASK:t2 > time = 1.
TASK:t2 > priority = 10.
TASK:t2 > task name = "EXIT".
TASK:t2 > knowledge source = "none".
    endwhen.

%
actions:

   message banner,
           welcome,
           banner.
%
```


ORIGINAL PAGE IS
OF POOR QUALITY