https://ntrs.nasa.gov/search.jsp?R=19950008644 2020-06-16T09:01:37+00:00Z

IN-20
31413
91 p

NASA Contractor Report 195388

Reusable Rocket Engine Turbopump Health Management System

Pamela Surko
Science Applications International Corporation
San Diego, California

October 1994

Prepared for
Lewis Research Center
Under Contract NAS3–25882

National Aeronautics and
Space Administration

(NASA-CR-195388) REUSABLE ROCKET ENGINE TURBOPUMP HEALTH MANAGEMENT SYSTEM Final Report (Science Applications International Corp.) 91 p

N95-15058

Unclas

G3/20 0031413

Table of Contents

Section 1 - Introduction

A health monitoring expert system software architecture has been developed to support condition-based health monitoring of rocket engines. Its first application is in the diagnosis decisions relating to the health of the high pressure oxidizer turbopump (HPOTP) of Space Shuttle Main Engine (SSME). The post test diagnostic system runs off-line, using as input the data recorded from hundreds of sensors, each running typically at rates of 25, 50, or .1 Hz. The system is invoked after a test has been completed, and produces an analysis and an organized graphical presentation of the data with important effects highlighted. The overall expert system architecture has been developed and documented so that expert modules analyzing other line replaceable units may easily be added. The architecture emphasizes modularity, reusability, and open system interfaces so that it may be used to analyze other engines as well.

Post Test Diagnostic System

The Post Test Diagnostic System (PTDS) aids engineers who are responsible for detecting and diagnosing engine anomalies from sensor data, by providing a consistent, fast first-pass data analysis. The analytical methods used by PTDS are modeled after these engineers' analysis strategies. The modular architecture has both procedural and non procedural knowledge-based components. This combination allows for the inclusion of conventional algorithms as well as heuristic expert information. Currently, this architecture has been implemented with the expert modules that analyze some aspects of HPOTP behavior.

Underlying Analysis Assumptions

All data being analyzed is time series data. Most cases involve examining the time dependent behavior of a single sensor, the difference between two sensors, or in some cases, a simple function of two to four sensors. Interaction between cooperating expert modules must incorporate time-dependence of features. Therefore the underlying strategies for handling and evaluating time series data must be carefully articulated, in order for new modules to be integrated into the system. In particular, a large system such as PTDS can grow too complex if care is not taken to provide only functionality that is essential. Therefore, a simple representation of time dependence that still has sufficient power for this domain was chosen. The system allows multiple

snapshots of the domain at different times, explicitly handling points in time (snapshot times) and time intervals. It also handles specific time dependence over an interval of an individual sensor trace.

Modularizing the Reasoning

In order to make much of the system reusable, we chose to divide the reasoning about sensor traces into two main categories. First, sensor traces are examined for "features" in their time series, such as peaks, spikes, level shifts, and the like. The number of features necessary to be identified for HPOTP analysis is relatively small. We anticipate that these features will be useful also for much of the reasoning for other LRU analysis. If other features are necessary for differing analyses, it is straightforward, given the architecture, to add modules for each new feature desired. After features have been identified and written to a database table, the rules perform the analysis work with the features, rather than with the individual data points or time series arrays. This split both manages the complexity of the analysis, and keeps the run time performance high. Since the features maintain their time dependence as parameters of the features, the reasoning process can treat time dependence with the full sophistication necessary. Features may be determined to be earlier or later than another, simultaneous, overlapping, or one feature may be encompassed in time by another. Features also have unique parameters such as height or width.

Section 2 - Architectural Framework

The Post Test Diagnostic System operates off line and with minimal user assistance. The system requires notification that a new test data set has arrived, and the input of several unit numbers of the engine Line Replaceable Units (LRU) being tested. The PTDS analyzes the data and prepares a results table for inspection by the data analysts. This process is typically conducted overnight and is ready for the data analyst in the morning.

Architectural Overview

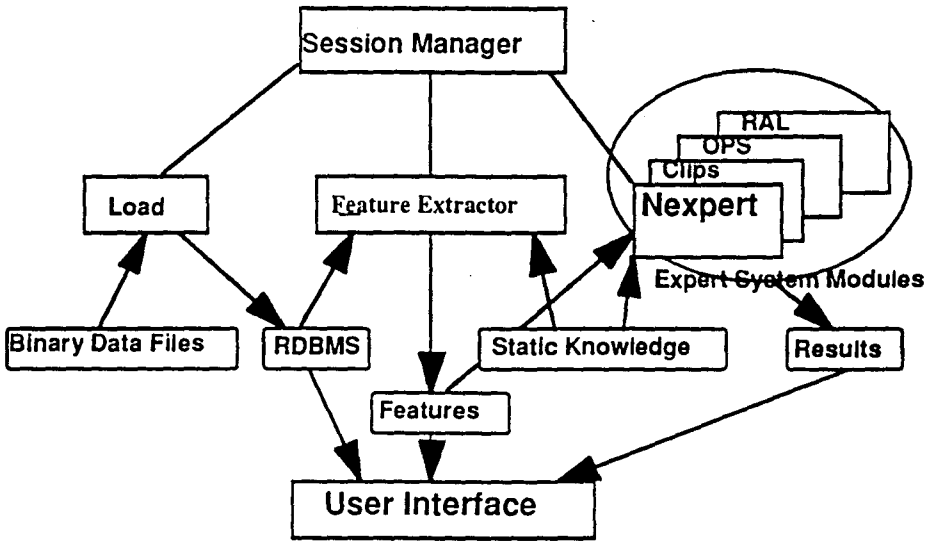

Figure 2-1: Architecture Overview

The data analysis process is implemented using the architecture shown in Figure 2-1. The controlling module is the "session manager". This UNIX process schedules and controls the loading of numerical data from unstructured binary files into the relational database management system (RDBMS), runs the feature extraction routines, and coordinates the expert modules. The expert modules reason using the features, static knowledge such as limits and expected noise levels contained in tables, and the knowledge contained in the rules. The user interface is a separate process,

which queries the results, features, and raw numerical data tables, to provide an intelligent display of the results.

Each segment of the system is described in detail in its own chapter. Following is a brief overview of each segment.

Session Manager Overview

The session manager controls the flow of data and the expert modules that analyze the data. It manages the execution of the expert modules by inspecting a resource table maintained in the RDBMS, to determine whether the prerequisites for an expert module have successfully been completed, and if so, then invoking the expert module as an independent UNIX process. This allows individual expert modules to be written in different languages, and gives individual expert module developers maximum flexibility.

A very simple mechanism for allowing modules to interact with each other has been implemented. A module can request information from other modules by writing to an RDBMS table which functions as the PTDS blackboard. If a request is written to the blackboard, the session manager re invokes the requested module, then re invokes the requester. All modules are responsible for inspecting the blackboard when they are invoked. To avoid deadlock, a module is required to be reinvoked before it can read the blackboard, and to avoid looping, each module can not be invoked more than twice, an arbitrary but satisfactory limit.

Database Schema Overview

The test data is managed in an **ssme_data** Ingres database, even though the official archiving of the numerical data is done elsewhere, using binary files. The RDBMS implementation strategy was chosen to ensure a robust system and to ease the porting of the system to the analysis of other engines. The features of data security, user privileges, checkpointing, and backup were provided by the commercial software and therefore could be eliminated from the custom software development.

The table **test_info** contains one record per test, and has all the data that appears once per test, such as test number, data and time of day of the test, LRU unit numbers being tested, and the length of the test.

The table **pid_info** contains one record per sensor, per test. Its fields store the sensor information for each test. A record contains the parameter identification number (PID), a description of what the PID is measuring on this test, the engineering

units, the sensor type, error bar information, one-second averages, and the sampling rate, typically 25, 50, or .1 Hz. The final table named for the test numbers contains the numerical data for all sensors in a particular test.

The table structure was chosen to optimize the speed of two types of queries: more important, the queries done by the user interface to retrieve data for display to the user; and second, the types of queries done by the feature extractor for its work. The feature extractor queries the database more often than does the user interface. This module operates in batch mode, with typical response times taking a few minutes. Since these types of queries are performed before the data analysts arrive, subsecond response times are not necessary, however, the addition of many more sensors may have an impact on future response times. The queries required by the user interface retrieve and graphically display data while the analyst waits, therefore display time in this case is crucial.

The time required to load the test data ranges from 20 minutes to more than an hour, depending on the length of the test firing. This loading time is slow, but since it is done overnight in batch mode, decreasing the load time was not necessary. A typical engine test totals 20-60 Mbytes, depending on the length of the test. Currently the system has 10-20 recent test firings resident in the database. The number of tests available on-line is limited by the space available on the database file system.

The RDBMS also manages feature tables, which store the intermediate results from curve fitting and other algorithmic analysis done on the numerical time series data; limits tables, which store limits used in feature detection and reasoning; and results tables, which contain the observations to be displayed to the user, and instructions for formatting the graphical displays to support those observations. The session manager has its own small database of tables containing instructions for managing the feature extraction process and tables of prerequisites for each process, allowing the session manager to be written without knowledge embedded into it of what order the various modules should be invoked, or what files or tables need to be present in order for them to function correctly. Since modules may be added throughout the lifetime of PTDS, the session manager must be as generic as possible.

Expert Modules Overview

One group of expert modules has currently been implemented. These modules analyze the seals, balance piston, and preburner pump bistability of the High Pressure Oxidizer TurboPump. The expert knowledge for the analyses is resident in three places: first, the list of which features to search for in which sensor traces; second, the tables storing the knowledge which does not change from test to test, used by the rules to define limits, allowed variability, expected noise levels, and the like; and third, the

knowledge embedded in the rules themselves. All numerical static knowledge is stored in RDBMS tables, rather than being hardcoded into rules or algorithmic code, to allow for ease of expansion and transfer. it is especially valuable to plan for maintainability in expert system development, since experts who have previously judged effects only by eye, using available hard copy graphs, may wish to try several values for bounding values.

Reasoning Strategies

The system reasons about whether two engine effects, as manifested in different sensors, may be related to a common cause or to each other, based on the time behavior of each occurrence. Also, PTDS reasons about the internal time structure of a single sensor signal by characterizing signals as "erratic", "spike", "peak", or "level_shift." Currently time dependence is implemented with one data structure (class), the feature, whose time-management properties are often simply start time and duration. Some features, such as asymmetric peaks in a time plot, have more complex time dependence, and the system carries the time of the maximum, and of both full-width-half-max points. Features are generated for all events of interest in the raw sensor data. Events are time-compared to other events using only the relations

- before
- simultaneous_with
- during
- overlapping.

Nearly all reasoning done by expert modules can be done with an appropriate set of features, rather than the full numerical data set. For efficiency's sake, the expert system does not reason directly about the individual data points, but about the features the experts have identified as important in the data. First, individual sensor traces are examined for expected behaviors, and a small set of useful curve-fitting routines are run to identify events in the time series data that are recorded as features. Only a small number of features are necessary to characterize most of the behaviors the expert system must analyze, such as "peak", "spike", "flat", "level shift", "different than" (comparing two traces) and "erratic behavior." The features are written to a database table and these, rather than raw sensor numerical data, are used as the symbols about which the expert modules reason. The expert module can then reason about features.

Two analysis strategies used by experts are used in the expert system. The current test is compared to results from a previous test to note unexplained differences, and the current test is also examined for evidence of a specified set of problems.

In searching for unexpected areas of change in the behavior of the turbopump

between the current test and a previous one, one would hope to compare the current test to the previous test running the same turbopump. This would be relatively simple, were it not for the fact that rarely are the operating conditions identical for two tests. The thrust profiles or the tank pressurization profiles may differ. The presence of a different low pressure turbopump, a different fuel turbopump or a different fuel/LOX mixture ratio will affect the operation of the HPOTP. In comparing two tests, each test is segmented into periods of constant thrust, by extracting level-change features on the thrust sensors, to identify the intervals at which the thrust was changing, and then identifying the time intervals between those thrust changes as periods of constant thrust. Each period of constant thrust is tagged with the actual value of thrust during that period, so that when comparing traces that should be the same within tolerances, comparisons are only done during periods of relative engine equilibrium, under similar thrust conditions. Thrust is the main driver of expected differences between tests. Rules have been designed but not yet implemented, to manage differences in tank pressurization as well.

An important piece of knowledge in doing test-to-test comparisons is the knowledge of which previous test should be used in the comparison. Each expert module makes its own determination of which previous test to use in current/previous comparisons. LRUs are swapped fairly often. For example, if the most recent previous test used the same HPOTP as the current test, but a different fuel turbopump, the system might choose that test for HPOTP comparison analysis, but choose the most recent test using the same fuel turbopump for a fuel turbopump analysis. In HPOTP analysis, the decision is made based on half a dozen criteria, with differing priorities. The system first searches for the most recent previous test on the same test stand, with the same engine, and the same HPOTP. Quite a number of rules handle the cases where not all the criteria are met. In order for the system to make best use of previous test data in doing these test-to-test comparisons, tests are analyzed in chronological order.

PTDS discards expected differences, and reports unexpected differences. This strategy reports unexplained new behavior of a pump without requiring knowledge of what caused that difference. Problems never before observed are detected, even though specific diagnostic rules for the anomaly are not present.

In the second reasoning strategy, using data from the current test only, several types of reasoning are done. One type exploits the limited redundancy available in the sensor data, by comparing both sensors to a previous test, if they differ, and assuming that if only one disagrees, that the physical quantity being measured agrees with the previous test and that one disagreeing sensor is faulty. If a parameter is sensed by only one sensor, then the dual hypotheses of actual physical change and sensor failure are made, unless other constraining evidence is available. The majority of the expert system rules treat the diagnosis of particular failure modes. For example, the balance piston module searches for various patterns and correlations in the traces from the two pressure sensors monitoring the pressures providing restoring forces for

the pump impeller. Spikes or level shifts in both pressures mean possible anomalies in the axial position of the impeller shaft. Spikes or level shifts in one pressure only imply different problems, and the relative sign of the spikes or level shifts give further clues as to what anomalies occurred. The system reports the unexpected feature it detected, and groups them where possible under a common root cause ("possible damage to a high pressure orifice").

Storing Expert System Results

For each anomaly, a record is written by the expert module detecting it to the results table of the ssme_data database. This table is accessed by the user interface when a user logs in to view the results of a test.

Using the Ingres table as a cache for results allows expert modules to be added to the system without requiring extensive changes to the user interface. The features for a test are also written to the features table by the feature extractor module, which runs before the expert modules.

When old tests are deleted from the database to release space, the features and results for that test are not deleted. Only the large numerical data table is deleted. This allows PTDS to return to older tests if they are appropriate for previous-test comparison. This does eliminate the ability to plot data or supporting data from the deleted tests, however.

User Interface Overview

The point-and-click color user interface allows the data analyst to view PTDS observations and supporting graphs. When the user invokes the system, the first window lists the tests residing in the system, and allows the user to sequentially choose the desired tests. An active engine diagram is offered, with engine LRUs highlighted if the expert module analyzing that LRU has detected anomalies. The user selects an LRU, highlighted or not, to see a display of the detailed schematic of that LRU annotated with PIDs and the observations associated with the LRU. If an anomaly is present in any PID, its label on the schematic is highlighted in color. By choosing a PID label or an anomaly line, the analyst displays the related time series data graphs. PTDS scales the data to highlight the important time segment, as well as displaying the supporting graphs often consulted by experts that show general engine conditions.

Section 3 - Session Manager

The session manager is the overall controller and job scheduler for the system as a whole. It is table driven where possible, so that additions and changes made to the system can be made easily using the RDBMS commands.

The session managers begins after an operator finishes entering data that is required by the PTDS, and which is not available in machine readable format, via use of a data entry screen. This data includes the test ID, and the numbers assigned to the LRU's on the engine.

The session manager utilizes tables that specify the tasks that need to be run and the resource required by that task. In many instances, a specific task cannot begin until previous tasks have completed. For example, the feature extractor task cannot begin until the Ingres tasks **`f_load`** and **`c_load`** have successfully completed loading the data.

By setting up a table that specifies only the prerequisites for each task, rather than a rigid ordering, we prevent the situation where a single minor error may unnecessarily stop the execution chain of the entire system. Instead, the system attempts to do all the tasks that may appropriately be done, given the previous tasks that completed successfully.

Database Schema

An overview of the database tables used by the session manager are shown in Figure 3-1. These tables are kept separate from the test data itself, in a separate database called **`session_mgr`**.

Resource_List

Id	name	displayable	whatif
0	data_arrival	Y	N
1	dbload	Y	N
2	features	Y	N
3	hpotp	Y	Y

Prerequisites

Id	pre1	pre2	pre3	pre4	pre5	pre6	pre7	pre8	pre9	pre10
1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1
2	1	-1	-1	-1	-1	-1	-1	-1	-1	-1
3	2	-1	-1	-1	-1	-1	-1	-1	-1	-1

Resource_Board

test_id	r0	r1	r2	r3	r4	r5	r6	r7	r8	r9
A10001	65	0	0	0	0	0	0	0	0	0
A10002	65	0	0	0	0	0	0	0	0	0
A10003	9	0	0	0	0	0	0	0	0	0

Job

module	test_id
dbload	A10003

Figure 3-1: Overview of database tables in the session_mgr database.

The **`Resource_List`** table shown in more detail in Figure 3-2, simply provides positive integer indices for each resource. Resources for a given module may be executable modules that must run to completion to provide the given module with necessary inputs.

Resource_List

id	name	displayable	whatif
0	data_arrival	Y	N
1	dbload	Y	N
2	features	Y	N
3	hpotp	Y	Y

Field Name	INGRES Variable Type	Comment
id	integer1	non-negative integer (0 - 127)
name	varchar(20)	resource name
displayable	char(1)	'Y' signifies resource appears in user interface resource status
whatif	char(1)	'Y' signifies resource appears in user interface whatif module

Figure 3-2: Schema of the Resource_List Table of the session_mgr database

The entry in the field **displayable,** of type char(1), is used to tell the user interface process which resources should be displayed, by test id, on the test status board. The test status board is shown in the top half of the initial window of the user interface. A **Y** indicates the resource should be displayed.

The **Whatif** field, also of type char(1), is used by the user interface to distinguish which resources are available to be run under the **Whatlf** module. A **Y** in this field means the resource is a **Whatlf** resource. (A **Whatlf** resource is one that should be rerun as part of the user's private hypothetical scenario, using hypothetical input data. See **Section 7** for further information about the what-if capability of the user interface).

Resource Prerequisites

Prerequisites

Id	pre1	pre2	pre3	pre4	pre5	pre6	pre7	pre8	pre9	pre10
1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1
2	1	-1	-1	-1	-1	-1	-1	-1	-1	-1
3	2	-1	-1	-1	-1	-1	-1	-1	-1	-1

Field Name	INGRES Variable Type	Comment
id	integer1	Index of executable resource, non-negative integer (0 - 127)
pre1 - pre10	integer1	index of prerequisites for resource id. Used prerequisites are non-negative integers (0-127). Unused prerequisites are negative integers (-128 through -1)

Figure 3-3: Schema of the Prerequisites table of session_mgr database

The **Prerequisites** table is shown in Figure 3-3. It provides the mechanism for specifying what resources are needed in order for a given module to be invoked. The first field is the index of the module whose prerequisites are being specified. This index corresponds to an id in the **Resource_List** table. The following fields are the indices of the resources required. If fewer than ten resources are required, the unneeded fields are filled with -1. The session manager only takes action for resources with nonnegative resource numbers.

Resource Activity Board

Resource_Board

test_id	r0	r1	r2	r3	r4	r5	r6	r7	r8	r9
A10001	65	0	0	0	0	0	0	0	0	0
A10002	65	0	0	0	0	0	0	0	0	0
A10003	9	0	0	0	0	0	0	0	0	0

Field Name	INGRES Variable Type	Comment
test_id	varchar(6)	string containing id of test
r0 - r9	integer4	resource variable encoding 15 resources per variable with two bits per resource: 0=Off, 1=On, 2=In progress or job error)

Figure 3-4: Schema of the Resource Board Table of the session_mgr database.

The **Resource_Board**, shown in Figure 3-4, is the session manager's private job status blackboard. This table is used by the session manager to track which modules have run correctly, which are currently in progress or have exited with an error, and which have not yet run. Since several tests may be in progress at the same time, each status log report has the test_id as its first field. When a resource becomes available (when a job has run and completed correctly, for example) its flag is changed by the session manager process to +1. When the session manager first invokes an executable, it changes the flag for that resource to -1. While the task is in progress, the flag remains set to -1. If it aborts or hangs up, the flag will stay at -1, indicating to the session manager that the resource is not yet available.

Although for clarity, Figure 3-4 seems to indicate that the number of resources is limited to only ten per task, the actual implementation (which is transparent to users) actually interrogates individual bits of integers, and thus the maximum number of resources that may be specified as required by a particular task is 127, which should be adequate for a realistically complex expert system.

The **Job** table is another table used internally by the session manager to form the command necessary to invoke the module. The Ingres table also preserves the job

stream status in the event of a machine crash or other failure. It is not set or used directly by programmers of individual expert modules.

Adding a New Module

As the expert system grows and evolves, there will frequently be the need to add a capability, such as a new expert system module, or some additional data acquisition or preprocessing capability. New capabilities must be written as executables that will work correctly when placed in any directory, and when invoked by the user "ingres". The executable must have one and only one argument, the testid, which the session manager will determine at run time and will use as the sole command line argument to the executable.

To add an executable into the expert system, perform the following steps:

- Place the executable (or a soft link to it) with appropriate permissions, in the appropriate directory.

- Login as the user "ingres" in order to have write permissions in the session manager's session_mgr database.

- Open the session_mgr database.

- Print out the contents of the Resource_list table, choose the next available integer as the index of your (new) executable, and place an entry with the index and the name of your executable in the table. If your executable requires new resources (such as input files) that are not already indexed in the system, give them indices in the Resource_List table as well.

- In the Prerequisites table, place an entry giving the index of your new module as the entry in the id field, and the indices of the resources your modules needs in order to run, in as many of the pre fields as are needed. Fill the rest of the 'pre' fields with -1.

Flow of Control

The session manager process itself is invoked each time the command **new_data** is issued. The **new_data** process queries for the test_id of the newly arrived test, and gathers the pretest information that must be entered by hand (LRU numbers).

The session manager then reads the resource_list, chooses the first item on the list where either all the prerequisites' flags are +1 ("on"), or if there are no prerequisites, sets the resource flag to "incomplete" (-1) for the module and invokes it. When the module completes with a normal exit, the session manager updates the resource flag to +1. While the module is executing, and if the module exits with an error, the flag remains at -1.

The session manager will continues until there are no more tasks that can be started. The session manager then quits.

Section 4 - Feature Extractor

Overview

The feature extractor program is at the root of SAIC's engine health management system for the space shuttle main engine (SSME). It provides the basic information on engine behavior necessary for operation of the expert modules. The expert modules use the sensor trace "features" reported by the feature extractor to reason about the health of an SSME component or assembly.

The feature extractor is currently capable of detecting the following generalized sensor trace features:

- **peaks** (All peaks or only the primary peak, where primary peak is defined as the peak having the greatest magnitude on the interval of interest)
- **spikes**
- **erratic behavior**
- **level shifts**
- **redline violations**

Sensor traces may also be statistically compared to determine the likelihood that they represent the "same" (two samples of data from the same parent distribution) or a different measurement or differ by a constant offset. This capability is provided by the feature extractor **different_than**. In addition to detecting the general features described above, the feature extractor is also capable of detecting more specific behaviors of the SSME such as changes in the net force exerted on the balance piston, and preburner pump bistability.

When invoked, the feature extractor reads a general command table that provides the program with basic information such as what type of features to look for and in what measurements (PIDs) to look for them. More specific information, such as the start and stop times of the search, and in the case of peaks, what type of model to fit to, are determined by the program at run time. By writing a set of general commands to the feature extractor, which are valid for all tests, consistent behavior is assured. In light of the consistent manner in which features are collected, the results may be used to accurately monitor the health of an SSME component.

Feature extraction is initiated by the session manager after the session manager has been notified of the arrival of new data. Both controller and facilities data for a test must be loaded prior to feature extraction. The first operation performed by the feature extractor is an analysis of the thrust profile for the test of interest. Periods of constant thrust are detected and classified by start time, stop time and percent thrust. This information is then used to provide parameters for each feature extraction module. Only features occluding during times of constant thrust are extracted. This provides protection against expected transients in the data which could manifest themselves as interesting features.

The feature extractor is designed to be run once for each analysis of an SSME test. A general command table, as referred to above, has been provided for the extraction of those features necessary for HPOTP analysis. No changes to this command table, or test specific setup is required. To extract features for the analysis of another module, appropriate commands must be appended to the command table. To alter any aspect of the program or command table between tests, aside from additions to the command table pertaining to additional modules, would make any further comparisons between tests invalid. The program needs only to be called with a different test id on the command line to provide features for another test.

Main Driver

The main driver of the feature extractor is responsible for building the array of run time commands based on the contents of the command table **feat_commands**, found in the **ssme_data** database, and the results of a thrust profile analysis. Each entry in the run time command array represents a call to a specific feature extraction module with parameters determined by the results of the thrust profile analysis for the test of interest and the contents of the command table. The main driver loops through this array of commands executing each named module with the specified parameters. The results of each feature search are appended to a temporary output file, found in **/tmp**, where a separate file is created for each type of feature. After all commands in the run time command array have been executed, the contents of these temporary files are loaded into the corresponding Ingres feature tables and the files deleted.

The columns in the command table are as shown below where each row represents a command to extract the named class of feature from the measurement.

- **expert** - This character string indicates the name of the expert module which requests the feature. The feature extractor runs only once per test so the features needed by all expert modules are extracted at the same time. This field is saved in the feature tables so that SQL queries may be issued for all features requested

for use by a certain expert module.

- **sensor** - This is a standardized string describing the measurement to be searched for the given class of feature. This string is used to look up the appropriate pid name which is an index into the data tables. For example, pid 63 is represented by the string "MCC Combustion Pressure, Average"
- **sensor postfix** - This indicates the use of either full sample data or one second average. The former is indicated by entering a lowercase "f" in this column while the latter is indicated by a lowercase "a". This field is used to qualify the contents of the "sensor" field which indicates which pid to operate on but does not specify whether to use sample rates of the raw data, or one-second averages computed by the PTDS.
- **modulename** - This is a string representing the feature extraction module to be called. The names of the available modules are as follows: DifferentThan, FindErraticBehavior, FindPeak, FindSpike, FindLevelShift, RedlineCheck, DeltaLevelShift, BalancePistonCompare, FindBistable, IsFlat.
- **starttime** - This character field contains a string indicating the time at which feature extraction is to start for this measurement. The time can be specified as an integer value or as one of the generic strings shown in the table below which represent times of interest common to all tests
- **endtime** - The contents of this field specify the time at which feature extraction is to end for the current measurement. Valid entries are the same as those described for **starttime**.

Valid entries for **starttime** and **endtime** fields

bot	Beginning of test data.
eot	End of test data.
cutoff	Engine cutoff time.
ts_eq	Time at which turbine seal equilibrium is reached.
lox_eq	Time at which LOX seal equilibrium is reached.

NOTE: Setting starttime = endtime indicates the special case where all periods of constant thrust are examined for the requested feature.

The fields **param1** through **param5** are character strings containing parameters specific to the named extraction module. Depending on the module, some, all or none of these fields may be used. In the event that a field is unused, its contents are irrelevant. Unused fields have been filled with an **X** for ease of inspection.

Thrust profile analysis of a test consists of detecting all periods of constant thrust over the entire range of data stored for **EHMS_ThrustPid**, which is currently defined to be the one-second average of pid 63 (pid "63A"). Pid 63 represents the engine's response to the commanded throttle value. Commanded throttle was originally chosen for this analysis as it is noise free, making thrust level changes easy to detect and the percent throttle value easy to determine. Currently, response to commanded throttle is used. It is less prone to falsely indicate as steady state, periods of changing thrust. It was found that in some cases the rise time of the engine to a throttle step was significant enough to cause the feature extractor to flag transient effects as features. In the interest of cutting down on the amount of features to be analyzed by the preprocessor (**HFILTER**) the pid used for thrust level analysis was changed from pid "287a" to pid "63a".

Each period of constant thrust detected has its start time adjusted to account for settling time, such that the resulting interval represents 95 percent of the original time segment (with a minimum allowance of one second and a maximum of three seconds). This operation moves the end points of the interval away from thrust transition times. This helps eliminate the effect of transients which may manifest themselves as features.

For each entry in the command table, where **starttime** equals **endtime**, a corresponding entry is made in the run time command array for each period of constant thrust. In this way only features which occur during periods of constant thrust are recorded, with each feature tagged by the thrust level at which it occurred. The user can override this operation and set predetermined start and end times for a feature search in a specific measurement, however if this time period spans more than one thrust level, any features found will carry thrust level ids of -999.

Thrust profile analysis is made automatically for each test. Each period of constant thrust is classified and stored in the Ingres table **feat_thrustleveldescrs**. This table contains the following fields:

module, test_id, sensor, start_time, end_time, offset, slope,

offset_sigma, slope_sigma, chi_square and thrust_level.

As mentioned above, **module** represents the name of the expert module which uses this feature in its analysis. The field **test_id** is the id string of the test from which the data used for feature extraction was drawn. The **sensor** field will contain the standard descriptor string corresponding to **EHMS_ThrustPid**. The **start_time** and **end_time** represent the start and end times of the period of constant thrust. The **offset** and **slope** represent the two parameters of a straight line fit to the data over the specified time range. The standard deviations on these parameters are given by **offset_sigma** and **slope_sigma**. The field **chi_square** is a measurement of how good the fit to the data was. The field **thrust_level** represents engine thrust level, as given by **offset** scaled to percent thrust.

The feature extractor may operate with either one second average or full sample rate data. Because many features happen over time scales of several seconds or longer, it is appropriate to fit them using one second average data. For each one second time bin there exists a standard deviation. These values are used to help track the movement of a measurement and provide the ability to correctly account for the noise in the data. Full sample data is used when necessary for determination of high speed features such as the 1/3 - 1/2 Hz oscillation in PBP bistability. When full sample data is used, only short intervals should be extracted from the data base to minimize memory use and maximize execution speed.

In the (hopefully rare) event of a severe run time error (such as the attempt to solve a singular system of equations) during the execution of any given command, the feature extractor main driver will log the complete text of the current command string in the file **BAD_COMMANDS**. This file will be found in the users current working directory. If **BAD_COMMANDS** already exists in the users current working directory, the command which produced the error will be appended at the end of the file. Note that the feature extractor does not halt execution in the event of an run time error. Any command which produces a severe run time error during a calculation, from which there can be no graceful recovery, causes a Unix signal 14 (**SIGALRM**). The handler routine assigned to **SIGALRM**, logs the current command in **BAD_COMMANDS** and executes a **longjmp** to cause feature extraction to resume with the next command found in the command table.

Feature Extraction Modules

The basis of all the feature extraction modules is the curvefitting routine **EHMS_MakeFit** which makes use of the Numerical Recipes routine "mrqmin". Source code for this routine, as well as a thorough discussion of its operation, can be found in section 14.4 of "Numerical Recipes in C, The Art of Scientific Computing"

(Cambridge University Press, 1988). **EHMS_MakeFit** can fit to any function that is differentiable with respect to the fitted parameters over the interval of interest. Each model for use in curve fitting exists as an independent subroutine. A pointer to the desired model for fitting is passed to the routine **EHMS_MakeFit**, indicating which model (functional form) to fit the data to. Currently models exist for an Nth order polynomial, a fast rising function with an exponential fall-off, as well as for a Gaussian (bell) curve.

Additional models can be added to the system by writing a short model routine based on the existing examples and making an entry in the routine **EHMS_GetNumBasisFuncs** to note how may fitted parameters are involved.

All feature results reported to the data base, irrespective of the routine that produced them, contain the fields: **module, testid, sensor, thrustlevel** and **start** and **end** time. The field **module** represents the name of the expert module which requested the feature. The field called **sensor** is the standardized descriptor string representing the name of the measurement. The **thrustlevel** field indicates the thrust level at which the feature occurred. The **start** and **end** time of the feature may fall into either of two categories depending upon the type of feature. Peaks, spikes, level shifts, and violations have a start and end time corresponding to the start and end time of the extracted feature. Erratic behavior, different than comparisons and bistability checks report start and end times which are equal to the start and end time passed to the extraction routine. This is reasonable as these modules check only for the presence of a condition on the specified interval, not for the presence of specific features.

For the collection of certain classes of features, limiting feature extraction to periods of constant thrust is inadequate to insure collection of valid features. Due to the methods employed in the feature extraction routines: **FindErraticBehaviour, FindLevelShift, DeltaLevelShift** and **FindSpike**, the effects of fuel tank repressurization must be accounted for. The feature classes noted above should only be collected during overlapping periods of constant thrust level and linear fuel tank repressurization/venting behavior.

Evidence of fuel tank repressurization and venting are searched for in one second average data for pid labeled, **'LPOTP Pump Discharge Pressure A'** (nominally pid #209). There is no direct measurement of this behavior so **'LPOTP Pump Discharge Pressure A'** is used as it was deemed the most sympathetic to the effects of interest. Repressurization and venting are indicated in pid 209 as linear periods which deviate from the usual horizontal trace, giving sections of the sensor data a sort of saw tooth effect. The algorithm which detects these periods, **EHMS_FindLinearLPOTPDischargePressure**, takes advantage of the observation that all sections of the trace, including those where venting or repressurization is taking place, exhibit a constant slope. The recursive routine

EHMS_FindInflectionPtsByLinearTrendRemoval is called by **EHMS_FindLinearLPOTPDischargePressure** to locate the start and end points of all periods of linear behavior in pid 209. By searching for features in other traces only during these linear periods (as well as periods of constant thrust) a reasonable assumption of steady state behavior can be made. The effects of venting and repressurization on other sympathetic pids are also negated and no longer cause erroneous periods of erratic behavior to be detected.

The routine **EHMS_FindInflectionPtsByLinearTrendRemoval** operates by first checking the data period (initially time zero to engine shutdown) to be sure the time span is greater than **EHMS_MinPeriodOfLinearTankPressure** seconds. The time slice must meet or exceed this requirement in order for the period to be valid for feature detection. Secondly any linear trend in the data is removed. This results in the start and end points being mapped to a magnitude of zero. The resulting data set is then searched for the point of maximum magnitude. If this maximum is greater than four times the standard deviation at that point, then the period is broken into two subintervals at the point of maximum magnitude and the routine recursively called for each subinterval. If the maximum magnitude is not greater than four times the standard deviation, then the interval is considered devoid of prominent peaks and is reported as a single continuous interval for the purposes of feature extraction. When all levels of **EHMS_FindInflectionPtsByLinearTrendRemoval** have returned due to reduction of subintervals to less than **EHMS_MinPeriodOfLinearTankPressure** seconds or lack of prominent peaks, the number of collected subintervals is returned as well as an array of start and end times for the periods of linear behavior.

While **EHMS_FindLinearLPOTPDischargePressure** does break out any periods displaying venting or repressurization effects, it makes no judgment about the data it generates. The algorithm was designed to negate the effects of venting and repressurization for the purposes of feature extraction. The feature extraction program does not need to know if the effects are occurring, it only needs to account for them.

Another source of potential error in feature extraction is produced by "bit_toggle". This effect is introduced by measuring a physical quantity with a sensor having limited dynamic range. The quantization error introduced into the measurements by using a quantizing step size as large as deviations of interest, produces a toggling effect seen in a graph of the collected data. In order to account for this type of error (which most often manifests itself as erroneous data spikes) the feature extractor searches for the smallest non-trivial step difference between two contiguous data points on the given interval. This value is taken to be the step size used by the sensor to quantize the analog measurements. By adjusting a calculated parameter up or down by half the step size, we may account for bit toggle error. One half the step size is equivalent to the maximum error in any given measurement. This toggle

extraction technique is used by the following feature extractor modules: **EHMS_FindErraticBehaviour, EHMS_DeltaDifferentThan** and **EHMS_FindSpike.**

Erratic (FindErraticBehaviour)

Command table inputs: expert, sensor, sensor_postfix, modulename, starttime, endtime, expected_sigma (param1)

This module is probably the simplest of the ten modules currently supported. Processing begins by comparing the number of points on the interval of interest to the number of parameters varied in a second order fit multiplied by the value of **EHMS_GoodFitFactor.** If the number of points is less than the result of the calculation described above, a first order fit is used instead of a second order. It was found that for short time intervals this technique produced better determinations of erratic behavior. After the fit has been made the standard deviation calculated for the fit is adjusted to account for bit toggle and then compared against an expected value, which is specified in the command table as **param1.** If the standard deviation for the fit exceeds this value, then the sensor trace is deemed to be erratic and information identifying which sensor and time period in which the condition was present is appended to the temporary file **feat_erratic**. The contents of the file are later loaded into the Ingres table **feat_erratic**.

The record for each entry will contain the following: **module, testid, sensor, start_time, end_time, thrust_level.**

Spike (FindSpike)

Command table inputs: expert, sensor, sensor_postfix, modulename, starttime, endtime

FindSpike is a variation on **FindErraticBehaviour.** It makes a fit to the data using a second order polynomial form, but for each data point on the interval a check is made for points which fall outside the limit defined by,

```
fitted_point +/- (EHMS_NumSigmasForSpike*fit_std_dev)
```

where **fit_std_dev** is the standard deviation calculated for the fit, adjusted for bit

toggles as shown below:

```
ftd_std_dev = fit_std_dev + (0.5 * step_size);
```

Any excursions outside this limit, which exists for no longer than **EHMS_SpikeWidth** seconds, are identified as spikes. The magnitude of the spike is reported as **magnitude**. The sign of **magnitude** is determined by the convention: **`raw_data[Index_of_peak] - fitted_point[Index_of_peak].`** The constants **EHMS_NumSigmasForSpike** and **EHMS_SpikeWidth** are currently defined to be 5.0 and 3 respectively.

The record for each entry will contain the following: **module, testId, sensor, start_time, end_time, magnitude, thrust_level**

Level Shift (FindLevelShift)

Command table inputs: expert, sensor, sensor_postfix, modulename, starttime, endtime

The purpose of **FindLevelShift** is to detect changes in a sensor trace from one constant value to another. It works by breaking the specified time period into sub periods of **EHMS_SubIntervalLength** seconds and making a separate first order polynomial fit to each period. The average and standard deviation of the constant offset terms of the fits are used to locate any level shifts. Any significant change in the data will manifest itself by a slope change in the set of fits to the data. A dramatic slope change will result in a line having a projection on the Y axis which far exceeds the average as determined from the multiple fits. Any excursions which exceed the average value plus or minus 3 times the standard deviation indicate the start of a level shift. Excursions are tracked until such time as they return to within acceptable limits or the period of interest is exhausted. Each level shift found is analyzed and the results appended to the database table **`Feat_levelshift`**.

The record for each entry will contain: **module, testid, sensor, starttime, endtime, last_magnitude, delta_magnitude, thrust_level.**

Peak (FindPeak)

Command table inputs: expert, sensor, sensor_postfix, modulename, starttime,

endtime, peak_set (param1), min_peak (param2), min_width (param3)

This module employs a technique where transitions in the data are identified by moving a tangent line over the data and noting the slope of the line. A line made up of three points is used, the center point being the "tangent" point. If the slope of this line exceeds a limit which allows for noise, then the time at the tangent point is marked as a transition time. Identification of a transition point is made by a check for slope > or < 3.0 times the standard deviation at the tangent point. The reasoning behind this is that the worst case slope which is still classified as noise will be ,

```
slope = (4 * sigma) / 2.0 = 2 * sigma
```

The slope of the tangent line must exceed 2 * sigma in order to be considered above the noise level in the data. Good results have been obtained using 3 * sigma.

The sign of the slope is also exploited for peak detection. Changes in sign indicate a cusp or peak in the sensor data. Only positive going peaks are detected, as indicated by a change in the sign of the slope from positive to negative. A peak is tracked from the first excursion outside the noise level to until either the measurement levels off, or begins to exhibit another peak. The detected peaks are then checked to make sure that they have a magnitude >= the minimum specified in the command table and that they have a width >= that specified in the command table. The data is then fitted to one of two models based on the position of the cusp relative to the start and end time of the peak feature. The first model is a fast rise with an exponential falloff. The second model is a gaussian curve . Those peaks with a chi-square per degree of freedom value of greater than **EHMS_ChiSquareFactor**, where **EHMS_ChiSquareFactor** is currently defined as 3.0, are discarded. The remaining peaks are then described and appended to the temporary table **peak.tmp**. If it was specified in the command table that only the primary peak be reported, then all others are discarded.

The entry for each record will include: **module, testid, sensor, peak_ht, taph** (time of peak height) , **fwhm** (magnitude of peak at half magnitude), **tafwhm1** (times where peak is at half magnitude), **tafwhm2, fit_type, chi_square, num_params,** the fitting parameters **param1, param2, param3, param4,** and **thrust_level.**

Different Than (DeltaDifferentThan)

Command table inputs: expert, sensor, sensor_postfix, modulename, starttime, endtime, compare_descrip (param1), polyorder (param2), num_comparison_sigmas (param3)

The **DifferentThan** module analyzes data from two sensors which may be drawn from the current test or the current test and a comparison test determined at run time. A first order polynomial fit is made to a composite data set where the data points and standard deviations are given by,

```
    delta_data[i] = data1[i] - data2[i];
    delta_sigmas[i] = sqrt((double) (sigmas1[i] * sigmas1[i] + sigmas2[i] *
sigmas2[i] ));
```

A fit which produces a line with little slope indicates that the two curves track each other well. The constant offset term produced by the fit indicates the distance maintained between the two data sets. A small value within acceptable error limits indicates that the two curves may be drawn from the same parent population. A constant offset term which exceeds acceptable error limits is indicative of a constant offset maintained between two curves. For the case where both fitted coefficients are outside error limits the two curves are determined to be "different".

After the fit is made to the composite data set, a probability measurement is made to determine if the two data set were drawn from the same parent population. This measurement is based on the Kolmogorov-Smirnov test. Small values of the measurement indicate that the cumulative distribution function of the first data set is significantly different from the second. This value, as well as the maximum step size found in either data set and the maximum average magnitude found on either data set, is used in the logic described below to determine if the two data sets can be considered to be "the same" or to differ by a constant offset.

If the result of the Kolmogorov-Smirnov test is greater than the value defined by **EHMS_SameAsProbability**, or three times the standard deviation calculated for the fit is less than half the maximum average magnitude, then a further check is made to determine if the higher order coefficients differ with acceptable error. If for each fitted parameter 1 through N (where **parm_sigmas** is the sigma for the corresponding parameter, adjusted to account for bit toggle),

```
param[i] - (num_comparison_sigmas * parm_sigmas[i])  >  0.0
   and
param[i] - (num_comparison_sigmas * parm_sigmas[i] )  <  0.0
```

then the two data sets are judged to be drawn from the same parent population. If the above is true for all parameters except the constant term, then the two data sets are said to differ by a constant offset. The offset value is given by **param[1],** and the standard deviation for this value is given by **parm_sigmas[1].** If none of the above is true and, all parameters 1 through N have statistically significant terms, then the two data sets are considered to be drawn from different parent populations.

The record written to the **different_than** table includes: **module, testid, sensor, compare_testid** (comparison test), **compare_sensor** (comparison sensor), **start_time, end_time,** the comparison statistics **chi_square1, chi_square2, prob, coefs_within_err_bars,** the offset flags **differ_by_offset, offset, offset_sigma,** and **thrust_level**

The two most common uses of the **DifferentThan** module are listed below, along with a short description of how to interpret the results produced.

1. Checking that redundant sensors are tracking each other within statistical limits.

To make this type of check, add an entry to the table **feat_commands** containing the command table inputs listed above. Setting **start_time** and **end_time** equal will cause the sensors to be compared for all periods of constant thrust. After the feature extractor has run, select the different than records for the current test **(test_id)** and the first pid listed in the command (sensor) from the table **feat_commands.** Inspect the **coefs_within_err_bars** field of each record. If any records are found which have a value of "False" for this field then the two sensor traces compared were found to differ statistically from each other. More specifically, the coefficients of the straight line fit to the difference data set were not within the tolerance set by **num_comparison_sigmas.** Further verification of the probability that the two data sets are not drawn from the same parent population can be determined by inspection of the field **chi_square** which represents the goodness of the fit to the difference data set and **prob** which is a measure of the actual probability that the two data sets were drawn from the same parent population. Small values of **prob** indicate that the two data sets are significantly different.

2. Checking for sensors which differ by a constant offset.

This check is handled in the same manner as the check described above but

involves inspection of the field **differ_by_offset**. If a value of "True" is found in this field, then the two data sets were found to track each other but with a constant offset. The value of this offset is found in the field **offset** The sign on this value is determined by the convention used to compute the difference data set used for the fit

```
delta_data[i] = data1[i] - data2[i];
```

A positive **offset** value indicates that the first data set listed in the table **(data1)** has values greater than the second **(data2)** by a constant term listed in the table field **offset.**

Redline Check (RedlineCheck)

Command table inputs: expert, sensor, sensor_postfix, expert system modulename, starttime, endtime, the comparison sensor compare_descrip (param1), check_type_str (param2), limit_type_str (param3)

The **RedlineCheck** module is used to check two pids to ensure that they stay within the limits defined for the given measurement. If an excursion beyond the limit defined is found, its duration is checked against a time limit value and decision logic is applied to determine if a redline has been violated.

Redline limit information is stored in the database table. All information needed for a redline check is extracted by the Redline Check module, the user need only specify (in the command table) the parameters check_type_str and limit_type_str. The parameter **check_type_str** is an enumerated type which may have one of the following three values: **both_pids, either_pid** or **difference.** Specifying **both_pids** causes the **Redline Check** module to look for instances where both pids exhibit an excursion beyond the specified limit at the same time. Choosing **either_pid** will indicate that only one pid must exceed the specified limit for a redline violation to occur. The last choice, **difference**, causes this module to look for redline violations in a composite data set made up of the point for point difference of the original two data sets. Note that the both_pids option should only be used with redundant pids as only one set of redline information is retrieved from the data base. Also when using the difference option a row of redline information must be present in the table **redline_info**, where the pid value is a string of the format **pid#- pid#**, which corresponds to the arguments for sensor and comparison sensor. The second parameter passed to this routine, **limit_type_str**, specifies whether the module should look for excursions below or above the lower or upper limit. These two options are specified by the

values lower and upper respectively.

The record written to the table, `feat_redlineviolations` will include: **module, testid, sensor, paired_sensor, violation_start, violation_end, check_type, limit_type, redline.**

Delta Level Shift (DeltaLevelShift)

Command table inputs: expert, sensor, sensor_postfix, the expert system modulename, starttime, endtime, compare_descrip (param1)

The **DeltaLevelShift** module serves as a front end to the **LevelShift** module. The purpose of this module is to produce a composite data set made up of the point for point difference between the two specified sensor data sets. A call is then made to the **LevelShift** module with the composite data set. Using this preprocessing allows the expert to look for level shifts in the difference between two pids. This is often useful in such cases as balance piston analysis where changes in the net force exerted on the balance piston can be detected by looking for level shifts in the data set comprised of pid 327 - pid 328.

Any features found by this module are reported to the same table used by the standard **Level Shift** module, `feat_levelshift`. The results of this module are distinguishable by an entry in the sensor column of the format **`pid#-pid#`**.

Balance Piston Comparison (BalancePistonCompare)

Command table inputs: expert, sensor, sensor_postfix, modulename, starttime, endtime, compare_descrip (param1), compare_test (param2), num_comparison_sigmas (param3)

The **BalancePistonCompare** module serves as a front end preprocessor to another standard module, **DifferentThan**. **BalancePistonCompare** produces a composite data set made up of the point for point difference between the two specified data sets. **BalancePistonCompare** creates two composite data sets, one with data drawn from the current test and one with data drawn from the comparison test.. A call is then made to the **DifferentThan** module with the composite data sets. The purpose of this specialized module is to look for changes in the net force exerted on the balance piston between tests at similar thrust levels.

Any features found by this module are reported to the same table used by the standard **DifferentThan** module, **feat_differentthan**. The results of this module are distinguishable by entries in the sensor and comparison sensor columns which, for **BalancePistonCompare,** are of the form: **pid#-pid#** which correspond to the sensor and comparison sensor.

Bistability (FindBistable)

Command table inputs: expert, sensor, sensor_postfix, expert system modulename, starttime, endtime

This special purpose module is intended to test for the presence of Preburner Pump Bistability in the SSME. This goal is accomplished by searching the pid defined by the variable **EHMS_BistablePid** for negative going spikes over each period of constant thrust having a thrust level of **EHMS_MinThustForBistablity** or lower. **EHMS_MinThustForBistablity** is presently defined as 65 percent thrust. The method used to detect spikes on the interval of interest is the same as that applied by the **Find Spike** module, with the exclusion of second order fitting capability.

If the number of spikes found on any given interval is greater than the number defined by the constant **EHMS_SpikeCountForBistability**, then that interval is flagged as containing an instance of **Preburner Pump Bistability.**

The record written to the table **feat_bistability** will include: **module, testid, sensor, fit_start, fit_end, thrust_level**

Conclusions

The feature extractor program provides a flexible, expandable system for the collection of important indicators of SSME performance. The system can quickly and easily be extended, or with only minor I/O changes, be applied to an entirely different problem . So long as the extraction modules are kept general, such as peak and spike, (features which may be found in almost any real world data set) the modules may be reused in other systems. Additional details on how to add to the feature extractor command table, and how to add a new feature extraction module are discussed in **Appendix A - Extending the System.**

Section 5 - Expert Modules

This section discusses the extent of the HPOTP knowledge acquired in Task 1 of the project, and which areas were chosen for implementation. It then delineates the knowledge acquired for the areas chosen for implementation.

Overview

Currently, there is one expert module running with the PTDS called the High Pressure Oxidizer TurboPump (HPOTP) module. Five areas were addressed by the HPOTP module: Preburner Boost Pump Bistability, Balance Cavity, and Primary, Intermediate, and Secondary Seals. All rules for the module were coded based on interviews with Marshall Space Flight Center expert analysts whose area of expertise was the HPOTP, and with former designers and testers of the HPOTP.

The basic purpose of the HPOTP module (as any other module in the PTDS) is to assist, not replace, the analysts by performing the more mundane and time consuming aspects of their data analysis. It will point out interesting aspects of the data, such as unexpected differences between the current and previous tests or unexpected structure in certain sensor traces, but it will be up to the analyst to perform the final failure analysis.

The mechanism chosen to point out these highlights is a list of short English language observations about unusual features in the data. It is known that most of the value of the expert system comes in giving interesting observations about the data (e.g.. an anomalous pressure rise seen in the time period $35 < t < 40$ sec, in the HPOTP primary seal drain pressure), rather than in attempting to diagnose exactly what caused the problem. The base cause of many anomalies is not possible to pin down, even by the most experienced experts. Sometimes even examining the dismantled pump does not yield an unambiguous explanation of an anomaly.

Therefore, although we intend to disambiguate postulates wherever possible, we expect that in some cases, several explanations will be plausible for an observed anomaly. The system will display all postulates if there is not sufficient reason to eliminate them. This will not greatly affect the value of the expert system analysis, since the main value is in correctly recognizing the interesting patterns in the data.

Although sensor validation will be done in a different module, a modest amount of analysis is done in this module as a placeholder until the sensor validation module is

added. In particular, although the capability exists in the feature extractor to check every sensor for erratic behavior or spikes, this is not done routinely in this module. In order to decide which of a redundant set of sensors to use for our reasoning, simple checks are made to ensure that redundant sensors give readings within expected errors. Rules are available for handling either sets of two or three sensors.

Bistability

The feature extractor module looks for bistability between engine start and shutdown. Any bistability features present for the current test are reported by a rule which examines the feature table **feat_instability**. The knowledge for bistability is contained in the feature extractor, and all the analytical work for determining bistability is described in the features chapter. The criteria specified by MSFC NASA experts are slightly different from those used by Rocketdyne, and hence in marginal situations, the two analyses give slightly differing results. MSFC experts have stated that this is not a problem, and that the criteria used in the expert feature extractor are a useful addition to the Rocketdyne analysis, as these expert criteria are more simply linked to directly measured quantities.

The data displayed in support of a bistability diagnosis includes the behavior of the engine controller. In cases of true bistability, the engine controller is asking for first more, then less, power from the HPOTP. This "sawtooth" behavior could be included in the determination of bistability, but is not at the present time. The sensor is displayed for the convenience of the viewer, however.

Balance Piston

The rules that deal with the balance piston pressures (327, 328) are concerned with features occurring between engine start and shutdown. There are two separate balance piston modules. The first module (**Appendix B Table 1**) has one rule that is fired when the difference of 327 and 328 is different from the current test to the previous test used for comparison. Since the feature extractor does the different than comparison during intervals of constant thrust and there may be many intervals with the same thrust level, the postulate will only report for the unique thrust levels. This module will not be executed if there is not a previous test to compare with in the database.

The second balance piston module is only done for the current test. The truth table in **Appendix B Table 2** is evaluated for features occurring simultaneously. A list of unique features start times is made for **Spike**, **LevelShift** and **DeltaLevel** involving balance piston pressures. Then the truth table is created internally for each

time in the list and evaluated. The '+' and '-' refers to the magnitude of the **Spike** or the offset of the **LevelShift** and a '0' means that the feature in question has no bearing on the postulate. Rows with two sets of conditions for one postulate means that if either set of conditions is true the rule declaring that postulate will fire.

Three Sensor Redunduncy

To maintain consistency throughout the rules it is necessary to choose one sensor to represent a group of sensors that measure the same quantity. The three sensor redundancy (951, 952, 953) has a possible complication in that one of the sensors may be missing. If this is the case one of the rules redpresented in **Appendix B Table A** will fire. If all sensors are present then the rules in **Appendix B Table B** will determine the choosen sensor (951A). This truth table gets set-up only once with the features occurring during mainstage, the longest interval of constant thrust with engine start time less than engine shutdown. An 'N' indicates that the feature extractor module **DifferentThan** did not detect a difference in the two sensors. Likewise a 'D' means there was a difference reported. Only one of these rules will evaluate to true. The choosen sensor is written to the database table **redund_sensor_choice**.

Two Sensor Redundancy

The rules in **Appendix B Table C** are used to choose a sensor to represent each of the two sensor redundancy packages (91, 92), (209, 210) and (211, 212). The features of interest occur at mainstage and if available from the previous test. If no previous test matches up to the current one, rules which only need features from the current test will choose the sensor. The entries 'No DIff', 'No Spike' and 'Not Erratic' mean that the feature was not detected. The choosen sensor is written to the database table **redund_sensor_choice** for each individual sensor package.

Similar Turbine Discharge Temperature Sensors

Features occurring during the mainstage interval of constant thrust for 233 and 234 are used to select one of these sensors to use throughout the expert. There are instances where the features present would normally fire more than one rule, like if both sensors were both erratic and showed spikes. Then the rule that looked for both erratic and spiked would fire plus the rules that looked for some of these features and do not inquire about the others would also fire. This module was developed (as was both redundant sensor modules) so that the highest priority rule whose left-hand conditions are satisfied would fire and those rules which inquire about the most

conditions are given highest proirity. The choosen sensor is then used in different rule modules where it is necessary to inquire about the turbine discharge temperature. The choosen sensor is written to the database table **redund_sensor_choice**.

Seals - Comparing Pressure and Temperature

This module focuses on whether changes in seal drain pressure can be detected in the drain temperature. This is done for the primary turbine (990, 1190) using rules displayed in **Appendix B Table 3**, secondary turbine (91A, 1188) using rules displayed in **Appendix B Table 6** and the primary pump (951A, 1187) using rules displayed in **Appendix B Table 8**. These rules look for the presence of erratic behavior and/or spikes exclusively during the mainstage thrust interval. The last postulate in each table will fire if any combination of erratic behavior or spikes is found in pressure and temperature. The three postulates in these tables are mutually exclusive.

Turbine Seals - Comparing Pressure to Previous Test

This module makes use of features from the previous test if available and reports if a change of more than a constant times sigma has occurred. As shown in **Appendix B Table 5**, peak height, time of peak and the full width at half max are the comparisons made with the pressures. Using the **DifferentThan** feature module the offset is compared at turbine seal equilibrium between the two tests. The absence of pressure peaks is also noted. The constant and tolerance values for the primary and secondary turbine pressures is retrieved from the **tolerances** database table.

Primary Turbine Pressure Peak and Equilibrium Checks

This module looks for pressure peak and equilibrium value shifts between the current and previous test as shown in **Appendix B Table 4**. **DifferentThan** features of interest are those that occur during the interval of constant thrust that includes the turbine seal equilibrium time or the first interval after that time. The turbine seal drain pressure peak occurs somewhere between start and engine shutdown. The sensor chosen that represents turbine discharge temperature (233A) for the current test is not necessarily the same sensor the expert choose when executed for the previous test, so the **DifferentThan** module will have previously done the following comparisons: 233 (current) vs. 233 (previous), 233 (current) vs. 234 (previous), and 234 (current) vs. 234 (previous).

Primary Pump Seal

This module does checks with the sensor chosen from the three sensor redundancy module (that picks 951A). The first postulate in **Appendix B Table 9** reports a non-flat primary pump seal drain pressure in the thermal equilibrium interval of constant thrust. The primary pump seal drain temperature is also checked for non-flatness but during the lox seal equilibrium interval of constant thrust. Since the feature extractor has no idea of knowing which sensor the expert will be choosing as the 951-952-953 package chosen sensor, the **DifferentThan** feature module has six comparisons to make to facilitate the check done for the third postulate. The fourth postulate requires a sign change between the current and previous test in the primary pump seal drain temperature during the lox seal equilibrium interval and the turbine discharge temperature during the seal thermal equilibrium interval.

Secondary Turbine Seal Cavity Pressure

The rules for **Appendix B Table 7** look for features occurring between start and engine shutdown. The first postulate will fire once if any combination of erratic behavior or spiking is present for both secondary turbine seal cavity pressure sensors (91 and 92). There is also a check to ensure that the time of peaks for the secondary turbine seal cavity pressure and primary turbine seal drain pressure is within a constant number of standard deviations.

Intermediate Seal Purge Pressure

The postulate in **Appendix B Table 10** reports any combination of erratic behavior and spiking for both intermediate seal purge pressure sensors (211 and 212). The time in the postulate is the smallest of the start times of the two features found. Only erratic behavior and spiking in 211 and 212 occurring between engine start and shutdown are considered.

Shutdown Checks

Appendix B Table 11 shows the postulates that require features occurring during the cooldown time after engine shutdown. Pump discharge pressure (90 and 190) and turbine discharge pressure (24) are checked to make sure they do not display erratic behavior or spiking. Since pump discharge pressure is measured by two sensors but

190 is a stale sensor, priority is given to the non-stale sensor to fire this rule.

Preburner Pump Bistability

Any bistability features present for the current test are reported by the rule described in **Appendix B Table 12**. The thrust level at which the bistability took place is output in the postulate string. The feature extractor module looks for bistability between engine start and shutdown.

Redline Violations

Appendix B Table 13 shows the redline violation postulate that gets fired for every redline feature for the current test. The sensor and violation time and whether the min or max limit was violated is output in the postulate string. Redline violations are looked for between engine start and shutdown.

Section 6 - Graphical User Interface

Introduction

The point-and-click color Graphical User Interface (GUI) system allows the data analyst to view the status of the observations made by the system for each processed test and graphical displays of supporting data.

The basic principles behind the use of windowing user interfaces, and the general manipulation of X-windows based interfaces has been covered in other documents and is reviewed in **Section 9 - Anomaly Database.**

Window hierarchy organization

When the user invokes the GUI, the first window lists tests processed by the system, and allows the user to choose the desired test. An active engine diagram is displayed and engine LRUs are highlighted if the expert module analyzing that LRU has postulates present. Any LRU, highlighted or not, may also be selected. When an LRU is selected, a display of the detailed schematic for that LRU appears, annotated with PIDs associated postulates. If an anomaly is present, PIDs associated with the anomaly are highlighted in red. By choosing either a PID label or a line containing an observation, the related time series data is displayed in graphical format. When a postulate is selected, all supporting data, as identified by the experts, is displayed for review.

During the interview process, the experts indicated a preference for three or four graphs per window that could be viewed without scrolling. With current usage of PV Wave software, the vertical scales are sometimes somewhat small for best human factors. It may be possible to design this shortcoming away in future releases of the user interface for other user communities.

Altering the User Interface

It is expected that the expert system will grow by the addition of more expert modules. Each of these modules will produce new results which will be placed into

the Ingres postulates table. The user must be guided to an appropriate grouping of these results. One way of doing this is to highlight the LRU being diagnosed on the initial main-engine screen, by placing a rectangular box around the affected LRU.

If the user chooses this box by clicking in it, then control passes to the new module. If the new module programmer would like to display a detailed drawing of the LRU, with the positions of each of the pids indicated, and allow the user to click on a pid to see graphical data associated with that pid, then the programmer may inspect the code in the HPOTP module that accomplishes this, and reuse much of it.

Main Screen Highlighting

To add the rectangle that will highlight the LRU on the main engine diagram screen, the programmer should look at the routines in **`EHM_boxes.c`**. No code needs to be changed to accommodate another highlighted region; it is only necessary to recompile the **`EHM_create.c`** module with a **-DMAKE_LRUS** flag set. This will allow access to the developer functions for sizing and moving a new box.

Pid Placement and Highlighting

In order to place active areas on a diagram, and to highlight them under programmer control (for example, to show a pid label in color if the pid has a postulate associated with it), as well as to design the box on the first screen, the code in **`EHM_create.c`** should be compiled with both **-DMAKE_LRUS** and **-DMAKE_PIDS** flags set. These flags will allow the programmer the functionality for changing boxes around LRU's on the main engine screen, and will also allow the functionality for creating and placing pids on a subsidiary LRU diagram. When this compilation has been done, running the user interface will cause the LRU screen to have a unique button **[Make Pids]** that is usually not available to the end user. Clicking this button allows the programmer to bring up a form that allows the moving of existing pids, and the specification of locations of new pids, and their names. These forms have a **[Save]** button on them. Clicking the **[Save]** button saves the data to a file which is used by the system to load in the rectangle locations for LRUs and the Pid information for the LRU screens.

In the EHMS resource file which globally is located in `/usr/lib/X11/app-defaults`, the rectangles file belongs to the **`boxFile`** resource and the pid information file belongs to the **`pid0File`** resource if it is associated with the **lru0Bitmap** picture, with different numbers associating with different bitmap files. (**`pid1File`** and **`lru1Bitmap`**, etc.)

To change an LRU diagram, one needs to have an **xbm** format diagram. Scanning a hard copy or using a drawing program may require the use of PBM or some other utility in order to get the diagram into **xbm** format. Note that the **xbm** format is black and white only. One can not use a color format for the LRU screens.

Once a diagram in **xbm** format exists, the programmer must make certain that the EHMS resource file has the appropriate resource to point to the file. The **boxFile** resource points to the data file containing the rectangles for the main engine screen. If there is a new LRU **xbm** diagram, and it is to be called when the user clicks on the main engine diagram inside the box specified by the second rectangle in the boxFile, then the programmer would set the **lrulBitmap** resource to the new **xbm** file. Note that resources are numbered beginning with the numeral 0, not 1. The pids to be placed on the new **xbm** diagram would be placed in the data file pointed to be the corresponding **pidl** file.

Section 7 - Database Design

Introduction

Numerical test data and ancillary data generated by the expert system and display system are maintained in a single Ingres database. The end users of the post test diagnostic system are not required to interact directly with the database through SQL commands. Instead, various user interface routines present a uniform appearance, and hide any unnecessary complexity from the end user.

Data is acquired for submission into the database management system from several sources. First, two types of files are transmitted from the remote location where the test occurred, and are transferred into a UNIX file system. These are the **.c** and **.f** files. These contain the time series data from "fast" and "slow" pids, as well as some general header information. Header information such as data of test, and test time of cutoff have some variability in format since the information is entered by hand. The year of the date, for example, is sometimes entered as one digit, sometimes two, and sometimes four. Header information specifying the frequency of time sampling for a particular pid is only applicable during the pre-cutoff period, and for some unspecified time after. At a point selected by a human monitor, pid data may be sampled at a lower rate than that specified in the header.

Another source of data is from hand-entry at the time the test is submitted into the post test diagnostic system. Some information needed by various expert modules, such as the serial numbers of the various LRUs being tested, the engine number, are not available in machine readable form at the time the test is submitted, and thus is entered by the administrator generating the request for the post test diagnostic system to analyze the data.

A third set of data in the Ingres database is generated by various modules of the post test diagnostic system. The feature extractor writes items ("tuples") into various tables, one per type of feature, for each feature discovered in the data. Feature tables' names begin with the prefix **feat_**. Expert modules analyze the features and write tuples into the postulates table, into "bad pids" and other tables such as **plot_info**, which details how best to display relevant graphical data. Details on these functionalities are found in Section 4 - Feature Extractor and Section 5 - Expert Modules.

A list of tables in the database may be seen by entering the interactive SQL editor by issuing the command

```
sql ssme_data
```

and then from the SQL editor, issuing the command

```
help \g
```

To see the fields of a particular table, for example the **test_info** table, one would issue the command

```
help test_info \g
```

To leave the SQL editor, issue the command

```
\q
```

which quits the session.

General Test Information

All information to be saved in the database which occurs once and only once per test is stored in the **test_info** table. This table is shown in Figure 7-1.

test_info

job_submit_date	test_id	date	engine#	cpids	fpids	Combustion_Devices	Controller	Nozzle	MCC	Main_injector	Powerhead	HPFTP	HPOTP	LPFTP	LPOTP	Engine_Shutdown
920129	A20531	910509	203500	B	B	combust	control	Nozzle	MCC	Injector	Powerhead	HPFTP	HPOTP	LPFTP	LPOTP	299.880
920129	A92345	910127	123	N	N	combust	control	Nozzle	MCC	Injector	Powerhead	HPFTP	HPOTP	LPFTP	LPOTP	123.450
920129	A92345	910128	123	N	N	combust	control	Nozzle	MCC	Injector	Powerhead	HPFTP	HPOTP	LPFTP	LPOTP	123.450
920129	A92345	910129	123	N	N	combust	control	Nozzle	MCC	Injector	Powerhead	HPFTP	HPOTP	LPFTP	LPOTP	123.450

Field Name	Variable Type	Comment
job_submit_date	integer4	yymmdd (date submitted to EHMS)
test_id	char(9)	unique test identifier
date	integer4	yymmdd (date of actual test)
engine#	integer4	unique engine part number
cpids	char(1)	
fpids	char(1)	
Combustion_Devices	varchar(20)	combusion devices part string
Controller	varchar(20)	controller part string
Nozzle	varchar(20)	nozzle part string
MCC	varchar(20)	MCC part string
Main_Injector	varchar(20)	main injector part string
Powerhead	varchar(20)	powerhead part string
HPFTP	varchar(20)	HPFTP part string
HPOTP	varchar(20)	HPOTP part string
LPFTP	varchar(20)	LPFTP part string
LPOTP	varchar(20)	LPOTP part string
Engine_Shutdown	float4	engine shutdown time (seconds)

Figure 7-1 test_info Database

Section 8 - Implementation

Hardware System

The PTDS system was developed on a Sun SparcStation 1, with an external SCSI hard disk drive of 1 Gbyte. The system is designed following UNIX open system standards, so may be easily ported to a variety of UNIX workstations, and users may make use of the X Window client server architecture to view results from a variety of UNIX workstations, X terminals or other monitors running the X Window software.

Commercial Software Packages and Languages

Three commercial software packages were used in the system.

- Nexpert Object, from Neuron Data was used to encode HPOTP expert system rules. At the time of this choice, (1988), CLIPS was eliminated from consideration by NASA because it was still relatively new and there was uncertainty about its performance characteristics. Nexpert Object has a graceful graphical programmers interface, and makes backward chaining systems relatively easy to encode. It has an embedded interface to Ingres.
- Ingres relational database management system. It was felt that the discipline and safety provided by a relational database management system would be a valuable addition to the system, since easy and dependable access to data was at the heart of the system. Ingres was already in use by NASA so it was selected for the PTDS for consistency. While relatively slow in loading test data (up to one hour) speed was not a major issue, since this process is performed overnight and will finish by the time the users need to review the results.
- PV~Wave display package from Precision Visuals. After evaluation of PV~Wave, DataViews and TAE+, PV~Wave was judged by end users to have most of the display features that they felt were important. This package handles the display of two dimensional (mainly time series) graphs.

The user interface package was written using the X11R4 and Motif Version 1.1 toolkits in Motif-compliant C code. Windows managed by PV~Wave were called from the basic user interface where required.

The feature extraction modules were written in C, for speed and access to C's mathematical libraries. The expert analysis for preburner pump bistability was also written in C.

Size of the System

Slightly more than 1000 Nexpert rules were written. These rules provide feature filtering to eliminate uninteresting features from further consideration, handle diagnosis of HPOTP balance piston problems, do the final HPOTP preburner pump bistability diagnosis, and analyze HPOTP seal problems.

Section 9 - Anomaly Database

.c.Introduction

The Anomaly Database of the SSME Post Test Diagnostic System gives rocket engine analysts an efficient, easy-to-use mechanism for tracking engine performance troubles. With the Anomaly Database, analyst experts may perform any of the following tasks appropriate to their responsibilities:

- after a test and data review, log and categorize any anomalies in SSME test data, along with expert assessments relating to the anomalies, actions taken, and the corroborating sensor data, if desired.
- retrieve data describing previously observed anomalies for analyzing patterns in engine performance
- retrieve all examples of classes of anomalies along with experts' determination of their causes for the purpose of training new analysts

The Anomaly Database is divided into two parts: the Ingres database itself which contains basic information about the type of anomaly observed along with sensor trace or traces which exhibit the anomaly most clearly; and a graphical user interface which provides a menu-driven front-end to the database so that the analyst does not need to remember details of how the data was stored or organized.

The database administrator is the only person to have direct access to Ingres commands. Other users will add, modify, retrieve, and delete records from the database via the graphical user interface according to their own particular access privileges. These access privileges are established by the database administrator and the system administrator.

The *System Requirements* section discusses the computational requirements for using an Anomaly Database system. Both hardware and commercial-off-the-shelf software requirements are included.

The *Overview* section contains a description of graphical user interfaces in general, how to use the mouse, manipulate windows, and invoke the Anomaly Database. This section demonstrates the general training required for the average user.

The *Organization* section gives an overview of the organization of the graphical user interface, and through it, the functional organization of the Anomaly Database. It includes a discussion about each functional group within the graphical user interface, a description of the edit modes available, and a description of the types of actions which the user takes in logging or retrieving anomaly data.

Browsing the Database provides a detailed description of how to browse through the anomaly database and retrieve appropriate data for viewing or printing This section covers the Read option provided by the graphical user interface.

Updating the Database provides a description of how to add, delete, or modify textual information and associated digital sensor data in the Anomaly Database. The Add, Modify and Delete options provided by the graphical user interface are made available only to those users with permission to alter the contents of the Anomaly Database.

The section *Administering the Database* provides a description of how the database administrator may access the underlying Ingres tables to alter the categories of anomalies, change user permissions, and the like.

System Requirements

The Anomaly Database resides on a Sun SparcStation 1 running SunOS 4.1.1. This machine is equipped with a large hard disk which also contains the database **ssme_data**. The Anomaly Database and the **ssme_data** database share some basic test information, and digital sensor data requested for entry into the Anomaly Database is fetched from **ssme_data** in order to ease the data entry burden on the users. The Anomaly Database requires the following Commercial Off-the-Shelf (COTS) software packages:

- Sun Operating System SunOS Version 4.1.1
- X Windows Version X11R4 or later
- Motif Version 1.1.1
- Ingres Version Release 6.4
- PV Wave Version 3.1

Overview

This section describes how users are oriented to working with graphical user interfaces (GUI's) in general and the Anomaly Database GUI in particular. Topics include inputting commands via the mouse, manipulating windows, and invoking the Anomaly Database program.

A Glance at Graphical User Interfaces

Graphical User Interfaces greatly extend the productivity of users by providing a seamless and uniform front-end to applications. Windows, buttons, and text guide the user effortlessly through an application without exposing the complexity of the underlying system.

A GUI typically provides what is called window-based mouse-driven functionality. A window provides the user with a direct link to an application. The application will display graphical objects, such as buttons, menus, text, and/or graphics, in the window. Some of the objects which appear in the window wait for the user to activate them whereupon they will signal the application that it must take the action associated with the object. For example, upon activation, the **[QUIT]** button will cause the application to stop running. A mouse provides a mechanism for selecting a window, and activating graphical objects within the window for the purpose of interacting with an application. Additional information on mouse features and functionality, and window attributes may be found in **Appendix C - GUI Features.**

Invoking the Anomaly Database

To invoke the Anomaly Database, users are instructed to first log onto a workstation or X-terminal, and (perhaps remotely) login to the machine containing the anomaly database. Once logged in, the display environment variable is set to the machine where the graphical user interface will appear. For example, if the machine the user is logged onto is named "sunxterm" and the host machine for the Anomaly Database is named "canada", then the user would perform the following operations:

```
rlogin canada
setenv DISPLAY sunxterm:0.0
```

The application is invoked as follows:

```
anomaly
```

The Organization of an Anomaly Database Window

The windows brought up by the GUI allow the user to specify what tasks need to be done, and to furnish the required information with a minimum of keyboard entry. The user fills in various fields on the window, edits and corrects them as necessary, and when satisfied, the user issues a command which then sends the request off to the database management system. (Before the "go" command is issued, the user's work as displayed on the screen can be thought of as local notes for an unsubmitted request.) When the requested information has been retrieved, the program will display it, usually on a window quite similar to that used to formulate the request.

Fields

Some areas of the screen are available for the user to enter material by typing or by choosing menu items. These are called "fields" or active areas. The graphical user interface has been designed so that if information is called for from the user, the field is active. In cases where information is not needed from the user, the area will not be active. Generally, when the program is providing information to the user, the area is not active. When the user is providing information to the program, the area is active.

Other Window Attributes

A top bar, just underneath the title bar, allows general commands. The leftmost button activates a short menu that either clears the screen of all user-entered material (useful when a user has finished looking at one record and is ready to request another), and the command to "quit" the program. The next button to the right issues the "go" command when clicked. This button submits a request to the database management system, based on the information on the screen.

All full-sized windows have a status line. This line provides information about what task the program is currently pursuing, or gives the status of the program. This line tells the user at a glance whether the program is waiting for information from the user, or is retrieving data, a process that can take several moments, if the amount to be retrieved is large.

Browsing the Database

Any request to retrieve information from a database management system is called a query. This user interface has been set up so that mentally, the user is requesting "show me all anomalies that have characteristics that match up with the fields I have filled in on the request screen." Any time a field is left blank, this means that there is no restriction placed upon the list of anomalies retrieved, based on that field. For

example, if all the fields were left blank and the **[GO]** button was clicked, the request would be for all anomalies in the database, a long list. If the **test id** field were filled with "A20531", then all anomalies that happened in that one particular test would be retrieved. (In English, the query would be "fetch all anomalies with test number A20531.") A user might wish to retrieve all green run violations. In this case, the user would move the cursor to the **[Spec Violation]** menu button, and choose **GREENRUN** to fetch all anomalies classified as green run violations.

In general, if one wishes to retrieve one particular anomaly, the simplest way to find it, without retrieving a large number of other anomalies that must be browsed through as well, is to fill in the fields that will place the most severe restriction on the list retrieved. Suppose, for example, one remembers that there was an interesting anomaly that happened sometime in 1990, involving a "start confirm" violation, on engine 0213. One also recalls that there are only a few "start confirm" violations in the database. If only the engine number and the year were specified, the query might retrieve a long list of candidate anomalies. Since there are only a few "start confirm" anomalies, however, that is the strongest criterion to use, and one should specify that criterion, perhaps without even bothering to add the engine number and date criteria. One could merely set the **[Spec Violation]** menu button to **ST CONFIRM** without typing anything in the date or engine number fields. and click **[GO]**. Whenever the program returns a list that is too long to examine conveniently, the user may return to the read window and add other restrictions on the list and issuing another **[GO]** command. This allows the user to browse a shorter list .

Preparing the Query

The user first clicks the **[Edit Mode]** menu button in the middle-left of the screen, below the status line, and sets it to **Read** if it is not already set.. If the screen is full of material from previous work, select **Clear** from the Options menu button in the upper left of the screen, which clears all fields. The status line will read: "Ready for a new query" when the screen is clear and is set in **Read** mode.

The user then enters facts about the anomalies to be retrieved by filling in data in fields that look indented, or by making menu choices in fields with menu buttons. Fields that are not active can not be used by the database management system to specify which anomalies to fetch. The exact entries in the various menus can be altered by the database administrator as the needs of the Anomaly database change. The contents of the user interface menus themselves are kept in Ingres tables for convenient administration.

Some specifics about the contents of each field in the Read (query) screen follow.

Test Number

The test number of the anomaly being searched for is one of the best restrictions to place on an anomaly search. The test number must be typed as a six character field, beginning with a capital A, followed by a five digit number. No dashes are allowed, and all test stands are now identified as A's. For example, A2-531 must be entered as A20531. Test B1-077, or 904-077, must be entered as A40077.

Test Date

If the date of the test is known, or even which month or year, this information may be used to shrink the list of anomalies retrieved by the request. The date must be entered as an eight character string composed of two month-digits, a slash, two day digits, another slash, and two year digits. To specify only part of the date, one may uses the character "?" to stand for "anything" in some parts of the date. For example, a query including "02/12/90" would retrieve only tests conducted on the 12th of February, 1990. A query with the date field filled in as "??/??/91" would retrieve all tests conducted in 1991. A query with the date field set to "06/??/90" would retrieve all tests in June of 1990. This capability to use "wild-card characters" allows some flexibility in specifying a date, when an approximate date of an anomalous test is known.

Engine Number

In order for a query to retrieve anomalies by matching on engine number, the engine number must be entered with the full four digits. If the leading digit is a zero, it must be entered. (A search for engine number "213" will not produce any anomalies, since the engine number has been entered as "0213" in the database and the system requires a character-by-character match.)

Anomaly Title - Location

This field is a menu button. The Location button allows one to specify whether the anomaly was in a sensor, a particular LRU, or a system problem. Specifying this field in a query allows one to choose, for example, to browse through all sensor problems, or all HPOTP problems. If **LRU** is chosen, then the rest of the query screen will change to allow the user to specify more about the LRU. All menu buttons start out labeled **BLANK** in **Read** mode, which means that no restrictions will be made on the anomalies retrieved, based on this field. If another menu entry is chosen, then it becomes the label on the menu. This allows a user to see what has been chosen.

Anomaly Title -Type

This field is also a menu button, and allows the location of the anomaly to become more specific. The contents of the menu in this field are context sensitive. They depend on the choice previously made by the user in the Location field. If this menu is not needed (for example, if the location field was left blank) it will not be active.

If the user chose **LRU** as the basic location of the anomaly, then a list of LRUs will be offered under this button. If the Location choice was **Sensor**, then this field would be used to specify which LRU the sensor was monitoring. (For example, to investigate how many times pressure sensors on the HPFTP have yielded anomalies, one could choose **Sensor** for location, and **HPFTP** for Type.)

Anomaly Title - Sensor Type

This menu allows the choice of all anomalies for a particular type of sensor; for example if one were interested in tracking all problems in pressure transducers on the high pressure fuel pump, one might pick **pressure** in this field, **LRU** for location, and **HPFTP** for Type.

Test Phase

This menu allows the selection of a test phase (prestart, mainstage, and the like)

Engine Flt/Dev

This menu allows users to specify whether they are interested in anomalies for flight engines or development engines. Leaving it in the **blank** condition means the anomalies retrieved are not limited to one or the other.

LRU Flt/Dev

This menu allows users to specify whether they are interested in anomalies for flight LRU's or for development LRUs. Leaving it in the **blank** condition means they do not wish to specify one or the other, and thus they would receive anomalies occurring in both flight and development LRU's.

Spec Violation

If the anomalies being sought were Spec Violations, the user may wish to specify one particular type of Spec Violation. Clicking on the menu button displays a list of Spec

Violation choices (such as **Greenrun, ICD, ICC, Max Qual, Min Qual,** etc.)

User Info

The database automatically records the logname of the person entering each record, and the date upon which the anomaly was entered into the database. These search fields are mainly for the convenience of users who are editing records in the database.

Submitting the Request

Once the request screen is satisfactory, the user clicks the **[Go]** button on the top bar. This submits the request. The Status line keeps the user up to date on the system's progress in performing the command, and changes the message as various steps are completed..

Error Messages

When one of the filled-in fields does not agree in format with that expected by the database management system, and the query can not be submitted, a window will display a suggestion to the user for editing one of the fields. To fix the problem, close the suggestion window containing the error message and the suggestion for correction, then move the cursor to the field that needs altering. To erase the field and retype, double click on the field and type over the old material. Or, place the cursor after the characters that need to be changed and backspace over them.

Examining the Results

When the user's request has been processed, the Status line informs the user how many anomalies satisfied the request. If the list contains more than one anomaly, a new window pops up labeled "Records that matched the read query".

The summary list of anomalies retrieved shows test number, anomaly location, type and problem. This summary information is intended to be just enough to help choose which items to examine in more detail.

A user scrolls through the list and clicks on an item to be examined in more detail. It is displayed by selecting the **[Load]** button on the upper left of the new window. The smaller window disappears and the primary window is loaded with information about the selected anomaly. Since there is not space on the screen to display all the textual material that may be part of the anomaly record, some information is stored "behind" buttons. For example, clicking on one of the **[Free Form Text]** buttons will pop up a window containing the paragraph(s) stored under that heading.

To view another anomaly from the list of anomalies retrieved by the previous query, select **[Read Selection]**, a new button which automatically appears directly under the **[Read]** button in the upper left part of the working screen, beneath the status line. This returns the "Records that matched the read query" window to the screen, where another anomaly may be chosen for further inspection.

Viewing Pid Data

If data was stored when the anomaly was entered, a user may graphically view the data that was chosen to illustrate the anomaly. The data available is that data that the person entering the anomaly felt was relevant and instructive. Not every anomaly has data stored with it. If the **[Data Stored]** button says "Yes", then clicking on **PIDs and Data** brings up an auxiliary window.

The initial data display shows the full time segment chosen for storage by the person entering the anomaly. If a user wishes to view a shorter time segment in order to examine minute details, the time axis may be changed by clicking in a time field and entering a different number. Most people choose initially to view the full time period in the database, however, in most usage scenarios, the user's first choice will be to select which pids to view. To view a pid, the user simply clicks on it in a **Select From** scrolled list, and the name and number of the chosen pid will automatically "hop over " from the **Select From** list to the **Use** list. When the user is satisfied with the **Use** list, the **[View]** button in the upper left of the window is clicked. This causes the database management system to retrieve the numerical data and display it in a scrolled graphical viewing area..

Depending on how many pids were chosen, not all of the graphs may be visible at once. To examine hidden graphs, a user simply "grabs" the scroll bar with the mouse and moves down in the graphical display.

Choosing many items means that the y axes may become somewhat compressed, in order to accommodate the large number of graphs. If the charts are too small for easy viewing, a user simply returns to the **Pids and Data** window and selects fewer pids into the **Use** list. Users may shift back and forth between the **Pids and Data** list and the graphical display at will, so that it is possible to choose to view several different displays of data for the same anomaly.

When the user is done viewing the data, selecting **Close** returns to the main window showing the anomaly.

Printing an Anomaly

To make a printed copy of one anomaly, or the whole list of anomalies satisfying the conditions posed by the query, the user clicks **Read Selection**, under the **[Read]**

button in the upper left of the screen, and moves the cursor into the "Records that matched the read query" window that appears in response. In this window, choosing **Print** yields a printed copy of the single anomaly currently being viewed, whose summary line appears in the bottom **Selected** box. Choosing **Print All** yields a printed copy of all the anomalies in the list of records that matched the read query. The printed reports contain all information stored for the query, except the graphical pid data. The full text of the long fields, hidden underneath buttons on the user interface screen, appear in the printed version, appropriately formatted..

Updating the Database

Adding new anomalies, and deleting or modifying existing ones are done with the Add, Delete and Modify commands. Most users of the database will not have write permission and will mainly be concerned with retrieving and analyzing existing anomaly data.

The ADD Command

The add command allows the user with appropriate permission to write in the database. A user may select **Add** from the [**Edit Mode**] menu button, to the left of the screen, beneath the status line. If a user does not have write permission, the **Add** command will be grayed out and inactive.

There are three types of fields that accept data entry: menus, short-entry fields, and free-form text windows. Menu fields behave similarly to **Read** window menus. The user clicks on a button, sees a list of possible choices, moves the cursor to the appropriate choice and clicks.

SHORT ENTRY FIELDS

Short entry fields accept a limited number of characters as input. These fields look like indented areas on the screen. In these fields, the user places the cursor upon the field area and clicks to activate the area. In some cases, the user interface software will perform some validity checks on the material typed into these fields before transmitting the database transaction to the database management system. If a problem is detected, the user interface software will bring up an informational window noting the problem and offering a suggestion for fixing it.

FREE FORM TEXT WINDOWS

Free form text windows appear as buttons on the screen. When the cursor is placed upon one of these buttons, and is clicked, a subsidiary window appears, with an

(empty, in this mode) text entry area. To activate the area, the user clicks on the area. It then becomes highlighted, showing it is active. The window behaves as most "WYSIWYG" (what-you-see-is-what-you-get, or "whizzy-wig") text editors. A user may type sentences with no carriage returns, and the software will automatically perform smart line-wraps, breaking sentences appropriately between words. The user may backspace over characters to delete them, or may mark a section of text for replacement by placing the cursor just in front of the first character to be replaced, holding the mouse button down while moving the cursor to mark the entire area to be replaced, and allowing the mouse button to come back up after the appropriate area has been marked. At this point, simply typing replacement text will overwrite the marked material. Or, if the material is not to be replaced with anything else, a single stroke on the delete key will eliminate all marked material.

If material is to be inserted, the cursor is placed in front of the character the material is to precede, and the user begins typing. The typed characters will appear to the left of the cursor. To append additional material to the end of material previously entered, the cursor is placed behind the last material entered, and text entry continued.

Formatting the Anomaly to be Added

The graphical user interface is designed to minimize typing wherever possible. This not only speeds anomaly entry and minimizes user input time, it lessens the opportunity for typing mistakes. Wherever data already exists in machine-readable form, the user interface will draw upon data already entered. Much of the material to be entered is new, however, and must be entered from the keyboard. Anomalies may be entered using the graphical user interface so long as the data for that test has already been stored in the Ingres ssme_data database. The user may begin by filling in whatever fields are easiest. Usually, data entry begins with the test_id, since the user interface's automatic data retrieval features require the test_id.

Test_id and other fixed fields

The test_id must be entered as a capital A followed by five digits. The database already has knowledge of the test date and correct engine number for this test, and the system will automatically retrieve it. The user may submit the add request with these fields left blank, and the system will fill them in.

Anomaly Location, Type, and Problem

These fields are menu fields. The user entering an anomaly should pick the most appropriate menu selections. If the Location was **Sensor,** an additional menu will appear to allow the user to specify what type of sensor gave the anomaly (flowmeter, pressure, temperature, etc.)

Power Leve

Any floating point number may be entered for percent power level. Numbers between 0.0 and 109.0 are most likely.

Test Phase, Engine Flt/Dev, LRU Flt/Dev

These menu fields must be filled in by the user. (Since engines and LRUs may change status from flight to development, the correct status must be entered by the user for each anomaly.)

Spec Violation

If the anomaly was a spec violation, the user should choose the **Yes** option in this menu field. When **Yes** is chosen, two more fields automatically appear. One is a menu button offering, as options, all types of spec violations. The other is a button that opens up a free form text window called **Violation and Criteria** which can accept up to 80 characters of commentary.

Assessment, Analysis Results, Actions Taken

These three fields are free form text entry fields. Clicking on the button opens a window suitable for text entry. Clicking in the text entry area activates it for use. The maximum length of each of these free form text fields is 1500 characters, which is about 25 lines of text.

When the text window's contents are satisfactory, the user closes the window by clicking its **[Close]** button.

Anomaly Time, Anomaly Duration

Two short entry fields are positioned following their labels. These fields accept floating point numbers. If identifying an anomaly time and/or duration is not appropriate for the anomaly being entered, these fields may be left blank.

Pid Info

Supporting data may be stored with the textual material, for future viewing, by selecting the menu choice **Yes** under **Pid Info**, and then selecting **[Pids and Data]**. This activates a new window, **PIDs and Data** which provides a menu listing all pids available in the ssme_data database for the test_id specified. This list is usually quite long, so it appears in a scrolling window. The scroll bar is "grabbed" by the user, by pressing down the middle mouse button and holding it down while moving the scroll bar up and down to change the viewing area onto the (long) list. When an interesting

area is in view, the middle mouse button is released. Pids are actually labeled in the database by an alphanumeric string, rather than with a simple number, and so appear on alphanumeric order in the list rather than numerical order. (91 and 911 may appear next to each other in the list.) Full sample data ids appear as numbers with no suffix, one-second average data appears as the same number as full sample, but with a lower case "a" appended. (Pid "63" is full sample data, pid "63A" is one second averaged data covering the same time periods.) If the database contains data taken at a rate slower than 1 Hz, this data will appear labeled with a suffix "s" for "slow", for example "63s".

The pid list scrolls both horizontally and vertically, since some of the labels are too long to display fully in the menu window. The labels associated with each pid number are those from the MSFC format file used to populate the Ingres database.

Viewing Data to Aid in the Storage Decision

The user interface allows data for selected Pids to be viewed or stored. This helps to determine the best pids and time slices to store to illustrate the anomaly being recorded. Prior to storing data consult with the database administrator for suggested limitations on how much full-sample or one-second average data should be stored. A user group's common practices will be guided by how much hard disk storage space is available.

Saving the Graphical Data

Once the graphical data has been selected, click **Save** to indicate that the choices of pid numbers and the time span of the data is correct. This will cause the auxiliary **PIDs and Data** window to close. The system will make a copy of the selected graphical data in the anomaly database tables from the **ssme_data** tables, since **ssme_data** tables are regularly purged for space reasons, and the anomaly data is expected to be available for much longer periods of usage.

User Information

The logname of the person entering the anomaly and the time of entry is automatically stored in the database. These fields are useful in editing anomalies. An anomaly administrator may recall some additional textual material to be added to an anomaly entered previously. In this situation, a very simple read request may be issued to retrieve, for example, all anomalies entered yesterday by myself, and editing of the desired anomaly may continue. It is also useful to know who composed the comments so that informal discussions may be initiated with the author.

Submitting the "Add" Request

When the screen is filled with all desired information to be added into the anomaly database, the author clicks the **[GO]** button in the top left corner. This will submit the addition. If by chance some typed material does not satisfy the formatting requirements of the database management system, an informational window will appear telling which field caused the problem offering a suggestion on changing its contents. No part of the record can be accepted by the database management system until all fields satisfy formatting requirements. When no informational windows appear after clicking the **[GO]** button, the user knows that the record has been successfully entered into the database.

The MODIFY Command

This command allows a modification of existing records in the database. When beginning with a cleared screen, the delete and modify menu choices in the **Edit Mode** box are grayed out and thus inactive. An anomaly must be selected to modify and bring it onto the screen before invoking the **Modify** command.

Identifying the anomaly for modification

Using the **Read** command as discussed above, the anomaly to be modified is located. It is not necessary to issue a query that will bring back only one anomaly. A query may be issued at any time to return a list of anomalies into the summary list screen containing the desired record. Individual records are selected for detailed viewing until the correct anomaly to be modified is found. When the anomaly to be modified is loaded on the screen, the user returns to the **Edit Mode** box and changes the menu choice from **Read** to **Modify**.

Changing Contents of Fields

Any field changed by the user on the screen will be changed in the database, once the **[GO]** command is issued. Until then the screen contents may be thought of as a scratch pad, where preparations are being made.

Short Entry Fields

Short entry fields look like indented areas on the screen. There are several ways to modify the contents of these fields. One method is to place the cursor in the field and double click the mouse. This "double-click" highlights the entire field and has the effect of erasing the field area so that the user may retype the area without backspacing over existing material. Another method is to place the cursor after

characters to be changed, backspace over them and retype them. Any other WYSIWYG editing functions may be used as well.

Menu Buttons

Menus are active and selectable, once the edit mode has been changed to **Modify**, so a user simply opens the menu by clicking on it and changing the choice from the one that was originally displayed on the button to the new choice. When the new choice is made by clicking, the menu button will display the new choice.

Long Text Fields

Clicking on the label of a long text field causes a window to be opened containing the text as it currently is stored in the database. Clicking in the window activates it for change, highlighting the text area. Changes to the module may be made with WYSIWYG editing features described earlier.

Replace the Modified Anomaly

To replace the old version of the anomaly with the new one as currently displayed on the screen, the **[GO]** button is selected. If the software detects difficulty with the formats of any of the changes, an informational window will pop up with a description of the problem and a suggestion for solving it. The change to the database can not be made unless the database management system's formatting requirements are satisfied completely. If an informational window pops up, the user reads the information, closes the window, makes the changes required in the anomaly data entered, and clicks **[GO]** again.

Since modifications are permanent changes to the database which make it difficult or impossible to get the old version back, a verification window will prompt for verification that the change is desired. This helps guard against accidental clicks on the **[GO]** button. After the user has verified that the changes are desired, the transaction will be performed. The status line provides information that the modification has been accepted and performed.

The DELETE Command

This command allows the administrator to delete existing anomaly records from the database. When beginning with a cleared screen, the user will notice that the **Delete** command is grayed out and inactive. This simply means that a **Read** command must be done first, to bring the anomaly into the work area prior to deletion.

Identifying the Anomaly to be Deleted

Using the **Read** command as described above, the anomaly to be deleted is brought into the work area. If the query returns more than one anomaly, an anomaly is selected from the summary screen. When the anomaly to be deleted is located, it is loaded onto the screen. In the **Edit Mode** box, the edit mode is changed from **Read** to **Delete** by clicking on **Delete**.

Deleting the Identified Anomaly

Once in **Delete** mode, the **[GO]** command on the top command bar is selected. A popup window then requires confirmation that deletion is truly intended. This guards against accidental clicks of the **[GO]** button, since it will be difficult or impossible to retrieve an anomaly that has been mistakenly deleted.

Administering the Database

Although basic system administration tools are being internally developed by NASA LeRC, and Ingres database administration tools are available as part of the commercial software acquired, the administrator of the anomaly database needs an understanding of the anomaly database design, in order to make good use of the tools. This section is intended to provide the necessary additional information. It assumes that the database administrator is already familiar with the basic Ingres database administration techniques and tools.

Table Design and Location

The anomaly database tables are located in the Ingres database **ssme_data.** This is the same Ingres database that contains the test data and results used by expert system modules. The anomaly tables are placed there to avoid time-consuming opening and closing of databases when users with "add" permission are entering new data.

The anomaly database stored its data separately from the data in the other data tables, but the user interface pulls information such as LRU numbers and test date directly from other tables in the **ssme_data** database during anomaly entry, to save the user extra typing, wherever possible. This both saves time and frustration for the user, and avoids opportunities for additional human error.

The anomaly database keeps separate copies of all information it needs, rather than pointers into the other tables, because we expect that the anomaly database will stay intact for a long time, whereas the administrator of the **ssme_data** database will

regularly delete or archive the large data files in the **ssme_data** data tables, because of space limitations. The **anom_info** table contains all the information about the anomaly except that required to support the display of relevant pid data, and the paragraphs of free form text that are allowed for analysis, assessment, and actions. The **anom_data_info** and **anom_data** tables contain the information required to store pid data with an anomaly if desired as well as the actual data.

The contents of the long free-form text entry fields are kept in a separate table, **anom_text**, in order to make a longer maximum field length possible. Each text entry field may have up to 1500 characters.

The contents of user interface menus that the administrator may at some point wish to change are also in tables; they are **anom_probdescr**, **anom_specviol**, **anom_testphases**, and **anom_sensortype**. The contents of these menus govern what the user adding new anomalies may add to the database, as well as what items a user may use to search the database for previously-entered anomalies.

Managing Database Size

General database administration will be done with tools provided by Ingres and LeRC. The anomaly database itself neither has a built-in limit on the size of the database, nor a watchdog process that monitors the size of the database or the disk space remaining. If the database grows too large, the administrator must archive and remove some material. Presumably, one prefers not to remove anomalies themselves. Therefore the best candidates for removal are probably supporting pid data sets. The administrator will decide, based on the needs of the user community, which pid data sets are least valuable. When they have been identified by anomaly number, the administrator may use the interactive SQL interface or the generic table editor provided by LeRC to remove records from **anom_data** and **anom_data_info**, keying on anomaly number. The administrator must also modify the general information record in **anomaly_info**, changing the start-time and stop-time fields to NULL. The user interface software needs these NULLs as the indication that there is no data stored with the anomaly.

Menu Item Tables

The four tables containing user interface menus are:

- anom_specviol
- anom_sensortype
- anom_testphases
- anom_probdescr

Anom_specviol

Anom_specviol contains the menu allowing the user to choose which type of spec violation occurred, if a previous menu indicated that a spec violation indeed occurred. This table has one field, named **violation**. At delivery time, this table contained:

```
GREENRUN
HOTFIRE
ICD
LCC
MAX QUAL
MIN QUAL
REDLINE
SOFTWARE
ST CONFIRM
```

Table 9-10... Contents of table anom_specviol

Anom_sensortype

Anom_sensortype contains the menu allowing the user to categorize which type of sensor the problem occurred in, if a previous menu indicated that a sensor fault occurred. It has one field, sensor. At delivery time this table contained:

```
Accelerometer
Control Flag
Drag On Cable
Flow Meter
Pressure
Pump Speed
Strain Gauge
Temperature
Valve Position
```

Table 9-11: Contents of anom_sensortype table

Anom_testphases

Anom_testphases contains the menu items designating which phase of the test the anomaly occurred in, if desired. This table has one field, phase. At delivery time,

this table contained:

```
Mainstage
Prestart
Shutdown
Start
Throttle
```

Table 9-12: Contents of table anom_testphases

Anom_probdescr

The most complex menu table is **anom_probdescr**, since it contains the contents of three hierarchically nested menus. **Anom_probdescr** has three fields; **Anomaly_location, Anomaly_type**, and **Anomaly_problem**, which correspond to the **Location, Type,** and **Problem** menus on the user interface. Entries in the **Anomaly_location** field list the possible areas of an anomaly. These areas can be broken down into **Types,** which are listed in the **Anomaly_type** field. These types can again be broken down into specific anomalies which are stored in the **Anomaly_problem** field. The current set of combinations for these three fields are shown in Table 9-13. Using the deliver-time contents of this table as an example, we can see that the first field contains the first menu choice, the second field contains the second, and the third field contains the "bottom", most specific, menu choice.

Anomaly Location	Anomaly Type	Anomaly Problem
LRU	Combustion Devices	MCC Fuel Leak
LRU	Combustion Devices	MCC Liner Cavity
LRU	Combustion Devices	MCC PC Delta
LRU	Combustion Devices	Main Injector Hot Gas Inj Pressure
LRU	Combustion Devices	Main Injector Delta P
LRU	Combustion Devices	OPB Pop
LRU	Combustion Devices	OPB Purge Pressure
LRU	Combustion Devices	Powerhead Hot Gas Leak
LRU	Combustion Devices	Powerhead LOX Leak
LRU	Combustion Devices	Powerhead Contamination
LRU	Controller	DCU-A Halt

LRU	Controller	DCU-B Halt
LRU	Controller	IEA Failure
LRU	Controller	OEA Failure
LRU	Controller	OEB Failure
LRU	HPFTP	HPFP Balance Cavity Pressure and Discharge Temp
LRU	HPFTP	HPFP Coolant Liner
LRU	HPFTP	HPFP Drain Line Leakage
LRU	HPFTP	HPFP Liftoff Seal Drain Line Freeze
LRU	HPFTP	HPFP Seal Leakage
LRU	HPFTP	HPFP Balance Cavity Pressure
LRU	HPFTP	HPFT Cavitation
LRU	HPFTP	HPFT Green Run Violation
LRU	HPFTP	HPFT Turbine Temp
LRU	HPOTP	Balance Cavity Pressure
LRU	HPOTP	Bearing Wear
LRU	HPOTP	Green Run Violation
LRU	HPOTP	HPOP Intermediate Seal Pressure
LRU	HPOTP	HPOP Primary Seal
LRU	HPOTP	HPOTP Bearing Failure
LRU	HPOTP	HPOTP Turbine Temperature
LRU	HPOTP	Late Breakaway
LRU	HPOTP	PBP Bi-stability
LRU	HPOTP	Rotor Grab
LRU	HPOTP	Secondary Seal Pressure
LRU	HPOTP	Speed Shift
LRU	LPFTP	Green Run Violation
LRU	LPFTP	LPF Duct Surge
LRU	LPFTP	Speed Shift

LRU	LPOTP	Cavitation
LRU	MCC	MCC Problem
LRU	Main Injector	Main Injector Problem
LRU	Nozzle	Nozzle Problem
LRU	Powerhead	Powerhead Problem
Sensor	Engine System Sensors	Engine Flowmeter Oscillations
Sensor	Engine System Sensors	Engine Flowmeter Shift
Sensor	Engine System Sensors	Engine Flowmeter Speed
Sensor	Engine System Sensors	MCC PC Shift
Sensor	Engine System Sensors	MC PC Spike
Sensor	Engine System Sensors	PC Delta Due to Inst
Sensor	Engine System Sensors	PC Drift
Sensor	Engine System Sensors	PC Shifts
Sensor	Engine System Sensors	Special Hex Redline
Sensor	Engine System Sensors	Turbine Temp
Sensor	Engine System Sensors	Turbine Tmp Drops Ec
Sensor	Facility Sensors	Facility Firex
Sensor	Facility Sensors	Facility Fuel Flowmeter Shift
Sensor	Facility Sensors	Facility OX Flowmeter Oscillations
Sensor	Facility Sensors	Facility OX Flowmeter Shift
Sensor	Facility Sensors	Low M/R
Sensor	LRU Sensors	Bad Connector
System	Engine System	Anomalous Frequence
System	Engine System	Electrical Lockup
System	Engine System	External Heat Exchanger Fuel Leak
System	Engine System	FPOV Position
System	Engine System	Fire
System	Engine System	Fuel Leak
System	Engine System	Fuel Turbine Temp

System	Engine System	HX Discharge Temperature
System	Engine System	High M/R
System	Engine System	Hydraulic Lockup
System	Engine System	KF Delta
System	Engine System	LOX Turbine Temp
System	Engine System	Low M/R
System	Engine System	M/R Shift
System	Engine System	OPOV Command Limiting
System	Engine System	Redline Violation
System	Engine System	Slow Start
System	Facility	Fac Differential Pressure
System	Facility	Facility Fire
System	Facility	Fuel Inlet Pressure
System	Facility	GN2 Temperature
System	Facility	He Internal Pressure
System	Facility	Input Electronics Failure
System	Facility	LOX Inlet Pressure
System	Valves/Hydraulics	AFV Valve Leakage
System	Valves/Hydraulics	CCV Closure
System	Valves/Hydraulics	GCV Pogo Precharge
System	Valves/Hydraulics	LOX Bld Valve Movement Failure
System	Valves/Hydraulics	MFV MFV Skin Temp
System	Valves/Hydraulics	PAV FPB Purge Pressure
System	Valves/Hydraulics	Pogo Precharge
System	Valves/Hydraulics	Pogo RIV Movement Failure

Table 9-13: Delivery Time Contents of anom_probdescr

Adding Menu Items

Adding menu items requires simply using the interactive SQL interface or the generic

table editor to insert a record into the appropriate table. Since the user interface builds its menus from the tables each time the interface is used, the new item will appear in the user interface the next time it is invoked after the administrator changes a menu table. The nested set of three menus stored in **anom_probdescr** require a bit more attention. Adding an item in the "bottom" menu (third one the user chooses) only requires adding one record. That record contains the choices made in the first and second menus, in order to get to the situation in the user interface where one chooses the new item being added. If the database administrator wishes to add or change an item in either the first or second menu, he or she must also specify what is supposed to appear in all the menus that would appear to the end user after that user chooses the item being added or modified. If one wished to add a new anomaly type (for example, a new LRU type called **New Thing**) in the case where the Anomaly Location was LRU, and if the possible Anomaly Problems with a **New Thing** are **Overheating, Rubbing, Icing** and **Seizing**, then one would add four records to the **anom_probdescr** table:

Anomaly Location	Anomaly Type	Anomaly Problem
LRU	New Thing	Overheating
LRU	New Thing	Rubbing
LRU	New Thing	Icing
LRU	New Thing	Seizing

Table 9-14: Additional records to add to anom_probdescr table, example.

Or, in a different hypothetical example, suppose one were to decide to split the second menu (**Anomaly Type**) entry **Valves/Hydraulics** into two separate categories, **Valves** and **Hydraulics**. One would have to examine the eight records containing **Valves/Hydraulics** in the second column, and decide where to place each of the third-choice menu items (**Anomaly Problems**). Some would need to be placed with **Valves**, some with **Hydraulics**.

Effect on the Database of Modifying Menus

The menus in the user interface provide the only access for the general user to the anomaly database. They are used both for entering data into the database, and for retrieving data. Therefore, if some data was entered into the database using a menu item which is subsequently deleted or changed, the user has no "vocabulary" for retrieving those items, by specifying the (missing) menu item in a query. The database contents must be inspected and perhaps altered, by the database administrator if menu items are deleted or modified. Adding a menu item does not require any action on the database itself, since the addition allows new kinds of records to be added to the database, but does not prevent access to any old ones.

If one deletes an item from a menu because it was not used, and there are no records

in the database that contain the field value that was deleted, then there are no consistency issues to deal with.

Other Administrative Issues

The administration shells written by LeRC will provide assistance in using the Ingres-provided tools for traditional database administration, such as backups, checkpointing, adding and deleting users, changing permissions for users, and the like. For a user to use the anomaly database, the administrator must grant two types of permission. First, the user must be made an Ingres user. We have granted universal permissions on all Ingres tables belonging to the anomaly database, but anyone using the database must have permission to use the **ssme_data** database. The main permissions barrier to unrestricted user access to the anomaly database is the UNIX group mechanism. The user interface checks to see if the user is a member of group "*anom_readonly*" or group "*anom_modify*". Therefore a new prospective user of the database must be given Ingres permission on the **ssme_data** database, and be made a member of one of these UNIX groups. The graphical user interface of the anomaly database looks up the permissions of the user running the interface. It grays out and deactivates commands such as **Add** which the user does not have permission to perform. When UNIX and database permissions are changed for a user, the graphical user interface requires no alteration. The new capabilities are available for the user on the next login.

Conclusion

The anomaly database system uses a standardized relational database management system, and keeps the contents of various user interface menus in Ingres tables. It is expected that this design will allow the database administrator to provide the correct amount of customizability without excessive complexity. The design goal was for sufficient flexibility that the database will enjoy years of active use, even in the continually-changing engineering and computing environment that its users and administrators face.

Appendix A - Extending the System

The following sections describe, in detail, how to add to the feature extractor command table in order to extract additional features from SSME data. The steps required in order to add a new feature extraction module are also discussed here.

Adding to the Command Table

The most efficient way to make additional entries in the command table is through use of the Ingres SQL script **load_commands.sql**. This script reads the ASCII file **commands.tmp** where each new command is represented by a line in the file with the fields separated by the "@" character. To run this script change directory to the space on the disk where both the load script and the ASCII command file reside and enter the Ingres interactive SQL terminal monitor with the command shown below

```
%sql  ssme_data
```

Once in the interactive SQL environment issue the following command

```
* \i  load_commands.sql
```

Ingres should issue a message notifying you of the number of rows (commands) read from the ASCII file.

Adding a new Feature Extraction Module

Like adding a curve fitting model, it is a simple matter to add an additional feature extraction module. Each module has a name identified by a **#define** statement in the file **EHMS_features.h**. By adding a name to this list and updating the **EHMS_NumFeatureExtractionModules** to the number of modules currently supported, a new feature extraction module can be made known to the system. Before the new module can be executed, an entry must also be made in the routine **EHMS_FeatureExtractor** where the module which corresponds to the defined name is called with the parameters (if any) listed in the base command table. Those modules already implemented should serve as an adequate example for implementing

a new capability. As an example, consider the case of adding the feature extraction module **IsSensorPegged**. The steps below indicate the operations which must be performed:

1. Insert the line below in the file **EHMS_features.h**

```
#define  IsSensorPegged n
```

where n is an integer equal to the number of the last module defined, plus one.

2. Increment the **#define** value of **EHMS_NumFeatureExtractionModules** by one.

3. Insert an additional "else if" condition in the routine **EHMS_ParseTableEntry** where the **module** string read from the command table is used to assign the corresponding integer from the **#define** list of module names. For the current example the addition shown below would be made to the else if construct.

```
   else if (!strcmp("IsSensorPegged",
extraction_module_str))
      *extraction_module = IsSensorPegged;
```

4. Make an additional entry in the switch construct found in the routine **EHMS_featureExtractor** for the new module. For our example this would read as follows:

```
   case IsSensorPegged:
      /* Get  and convert the types of  all params in the
command */
      /* call the routine which corresponds to
IsSensorPegged */
   break;
```

Appendix B - Tables

Table A. Three Sensor Redundancy (Missing Sensor)

Sensor i Present	Sensor j Present	Sensor k Present	Use	Postulate
Yes		No	i	Sensor 'j or k' is missing, using 'i' to continue the analysis.
Yes	No			
No	Yes		j	Sensor 'i or k' is missing, using 'j' to continue the analysis.
	Yes	No		
No		Yes	k	Sensor 'i or j' is missing, using 'k' to continue the analysis.
	No	Yes		

Table B. Three Sensor Redundancy

Compare i j	Compare j k	Compare k i	Use	Postulate
N	N	N	i	Sensors i, j and k agree, using i for convenience.
N	D	D	i	Seen beginning at t = __, sensor k disagrees with i and j, using i for convenience.
D	D	N	i	Seen beginning at t = __, sensor j disagrees with i and k, using i for convenience.
D	N	D	j	Seen beginning at t = __, sensor i disgrees with j and k, using j for convenience.
N	N	D	i	Seen beginning at t = __, sensors i, j and k inconsistent, using i to continue the analysis.
N	D	N	i	
D	N	N	i	
D	D	D	i	

Table C. Two Sensor Redundancy

Feature							Use	Postulate
Diff i, j (current) (current) Diff i, j (current) (current) Diff i, j (current) (current) Diff i, j (current) (current)	Erratic i Erratic i	 Spike i Spike i	Erratic j Erratic j	Spike j Spike j	No DIff i, i, (current), (previous) No DIff i, i, (current), (previous) No DIff i, i, (current), (previous) No DIff i, i, (current), (previous)	Diff j, j (current), (previous) Diff j, j (current), (previous) Diff j, j (current), (previous) Diff j, j (current), (previous)	i	Seen beginning at t = __, sensors'i' and 'j' show jitter and 'j' does not track previous test. Using 'i' as the best value.
Diff i, j (current) (current) Diff i, j (current) (current)	Not Erratic i Not Erratic i	No Spike i No Spike i	Erratic j	Spike j			i	Seen beginning at t = __, sensors'i' and 'j' agree. Sensor 'j' shows jitter while 'i'does not. Using 'i' as best value.
Diff i, j (current) (current) Diff i, j (current) (current)	Erratic i	Spike i	Not Erratic j Not Erratic j	No Spike j No Spike j			j	Seen beginning at t = __, sensors 'j' and 'i' agree. Sensor 'i' shows jitter while 'j'does not. Using 'j' as best value.
Diff i, j (current) (current)	Not Erratic i	No Spike i	Not Erratic j	No Spike j	DIff i, i, (current), (previous)	No Diff j, j (current), (previous)	j	Seen beginning at t = __, sensors 'i' and 'j' disagree. 'j' tracks previous test. Using 'j' as the best value.

Table C. Two Sensor Redundancy (continued)

Feature							Use	Postulate
Diff i, j (current) (current)	Not Erratic i	No Spike i	Not Erratic j	No Spike j	No DIff i, i (current), (previous)	Diff j, j (current), (previous)	i	Seen beginning at t = __, sensors 'i' and 'j' disagree. 'i' tracks previous test. Using 'j' as the best value.
Diff i, j (current) (current) Diff i, j (current) (current) Diff i, j (current) (current) Diff i, j (current) (current)	Erratic i Erratic i	 Spike i Spike i	Erratic j Erratic j	 Spike j Spike j	DIff i, i, (current), (previous) DIff i, i, (current), (previous) DIff i, i, (current), (previous) DIff i, i, (current), (previous)	No Diff j, j (current), (previous) No Diff j, j (current), (previous) No Diff j, j (current), (previous) No Diff j, j (current), (previous)	i	Seen beginning at t = __, sensors'i' and 'j' show jitter, 'i' does not track previous test. Using 'j' as the best value.
No Diff i, j (current) (current) No Diff i,j (current) (current)	Erratic i	Spike i	Not Erratic j Not Erratic j	No Spike j No Spike j			j	Seen beginning at t = __, sensors 'j' and 'i' disagree. Sensor 'i' shows jitter while 'j'does not. Using 'j' as best value.

Table C. Two Sensor Redundancy (continued)

Feature							Use	Postulate
No Diff i, j (current) (current)	Erratic i Erratic i		Erratic j	Spike j				Seen beginning at t = __, sensors'i' and 'j' both show spikes or erratic behavior. Disregard all results using these sensors.
No Diff i, j (current) (current)		Spike i Spike i	Erratic j	Spike j				
No Diff i, j (current) (current)								
No Diff i, j (current) , (current)								
No Diff i,j (current) (current)	Not Erratic i	No Spike i	Not Erratic j	No Spike j			j	Sensors 'i' and 'j' agree, using 'i' for convenience.
No Diff i, j (current) (current) No Diff i, j (current) (current)	Not Erratic i Not Erratic i	No Spike i No Spike i	Erratic j	Spike j			i	Seen beginning at t = __, sensors'i' and 'j' agree. Sensor 'j' shows jitter while 'i'does not. Using 'i' as best value.

Table 1. Balance Piston Pressure Difference Between Two Tests

Feature	Postulate
Delta DifferentThan 327-328, (current), (previous)	First noticed at t = __, the difference (327 - 328) is different at thrust level __, between this test and the previous.

Table 2. Balance Piston Truth Table

Spike 327	Spike 328	Level Shift 327	Level Shift 328	DeltaLevel Shift 327 - 328	Postulate
+-	0	0	0	0	Seen at t = __, spike seen in sensor 327 only, with no change in steady state pressures or pressure difference. Possible sensor or omni seal anomaly. No real rotor motion.
0	+-	0	0	0	
		+-	0	+-	Seen at t = __, level shift seen in sensor 327 only. Possible sensor problem omni seal leakage or cup washer problem. No real rotor motion.
		0	+-	+-	
+	-	0	0	0	Seen at t = __, possible HPOTP balance piston momentary orifice change or momentary anomalous rotor motion.
-	+	0	0	0	
		+	-	+-	Seen at t = __, possible HPOTP anomalous rotor motion.
		-	+	+-	
		+	+	0	Seen at t = __, possible HPOTP balance piston orifice damage.
		-	-	0	
		+	+	+-	Seen at t = __, HPOTP balance piston anomaly may be unusual rotor motion or orifice change. Same-sign level shifts in both pressures, with change in pressure difference as well.
		-	-	+-	
		+-	0	0	Seen at t = __, statistically significant change in 327 but not in difference (327 - 328)
		0	+-	0	Seen at t = __, statistically significant change in 328 but not in difference (327 - 328)
		0	0	+-	Seen at t = __, statistically significant change in difference (327-328) but not in individual sensors

Table 3. Primary Turbine Seal - Comparing Pressure and Temperature

Feature	Feature	Feature	Postulate
Erratic 990	Not Erratic 1190	No Spike 1190	Seen beginning at t = __, HPOTP erratic primary turbine seal drain pressure may indicate sensor problem, seal anomaly, or vibration. No effect seen in drain temperature.
Erratic 1190	Not Erratic 990	No Spike 990	Seen beginning at t = __, HPOTP erratic primary turbine seal drain temperature may indicate sensor problem, seal anomaly, or vibration. No effect seen in drain pressure.
Erratic 990 Spike 990 Erratic 990 Spike 990	Erratic 1190 Erratic 1190 Spike 1190 Spike 1190		Seen beginning at t = __, HPOTP shows jitter in both primary turbine seal drain pressure and temperature. Possible seal anomaly or vibration.

Table 4. Primary Turbine Seal Pressure Peak and Equilibrium Checks

Feature	Feature	Feature	Criteria	Postulate
Diff 233A, (current), (previous) $offset_1$	Peak 990, (current), (previous) peak_ht	Diff 990, (current), (previous) $offset_2$	$peak\_ht*offset_1>0$ $peak\_ht*offset_2<0$ at turbine seal equilibrium	HPOTP primary turbine seal drain pressure peak and equilibrium values have shifted in opposite directions compared to previous test. Consistent with change in turbine discharge temperature.
Diff 233A, (current), (previous) $offset_1$	Peak 990, (current), (previous) peak_ht		$peak\_ht*offset_1<0$ at turbine seal equilibrium	HPOTP primary turbine seal drain pressure peak shift compared to previous test inconsistent with change in turbine discharge temperature.
Diff 233A, (current), (previous) $offset_1$		Diff 990, (current), (previous) $offset_2$	$peak\_ht*offset_2>0$ at turbine seal equilibrium	HPOTP primary turbine seal drain pressure equilibrium value has shifted compared to previous test. inconsistent with change in turbine discharge temperature.
	Peak 990, (current), (previous) peak_ht	Diff 990, (current), (previous) $offset_2$	$offset_1*offset_2>0$ at turbine seal equilibrium	HPOTP primary turbine seal drain pressure has shifted from previous test both in peak and equilibrium value. May be change in seal clearance or sensor calibration.
No Diff 233A, (current), (previous) $offset_1$	Peak 990, (current), (previous) peak_ht	Diff 990, (current), (previous) $offset_2$	$offset_1*offset_2<0$ at turbine seal equilibrium	HPOTP primary turbine seal drain pressure peak and equilibrium value have shifted in opposite directions compared to previous test. No apparent change in turbine discharge temperature.

Table 5. Turbine Seals - Comparing to Previous Test

Feature	Feature	Compare	Postulate
NoPeak 990, (current)			Current test HPOTP primary turbine seal drain pressure peak missing.
No Peak 990, (previous)			Previous test HPOTP primary turbine seal drain pressure peak missing, therefore no comparisons done.
Peak 990, (current)	Peak 990, (previous)	peak height	HPOTP primary turbine seal drain pressure peak height has changed by more than 'cut_width' * sigma between previous test and this test.
Peak 990, (current)	Peak 990, (previous)	time at peak height	HPOTP primary turbine seal drain pressure peak is earlier or later than previous test by more than 'cut_width' * sigma.
Peak 990, (current)	Peak 990, (previous)	full width at half max	HPOTP primary turbine seal drain pressure peak width is wider or narrower than previous test by more than 'cut_width' * sigma.
Diff 990, (current), (previous)		pressures at turbine seal equilibrium	HPOTP primary turbine seal drain pressure at thermal equilibrium is more than 'cut_width' * sigma different from previous test.
No Peak 91A (current)			Current test HPOTP secondary turbine seal cavity pressure peak missing.
No Peak 91A (previous)			Previous test HPOTP secondary turbine seal cavity pressure peak missing, therefore no comparisons done.
Peak 91A, (current)	Peak 91A, (previous)	peak height	HPOTP secondary turbine seal cavity pressure peak height has changed by more than 'cut_width' * sigma between previous test and this test.
Peak 91A, (current)	Peak 91A, (previous)	time at peak height	HPOTP secondary turbine seal cavity pressure peak is earlier or later than previous test by more than 'cut_width' * sigma.
Peak 91A, (current)	Peak 91A, (previous)	full width at half max	HPOTP secondary turbine seal cavity pressure peak width is wider or narrower than previous test by more than 'cut_width' * sigma.
Diff 91A, (current), (previous)		pressures at turbine seal equilibrium	HPOTP secondary turbine cavity pressure at thermal equilibrium is more than 'cut_width' * sigma different from previous test.

Table 6. Secondary Turbine Seal - Comparing Pressure and Temperature

Feature	Feature	Feature	Postulate
Erratic 91A	Not Erratic 1188	No Spike 1188	Seen beginning at t = __, HPOTP erratic secondary turbine seal drain pressure may indicate sensor problem, seal anomaly, or vibration. No effect seen in drain temperature.
Erratic 1188	Not Erratic 91A	No Spike 91A	Seen beginning at t = __, HPOTP erratic secondary turbine seal drain temperature may indicate sensor problem, seal anomaly, or vibration. No effect seen in drain pressure.
Erratic 91A Spike 91A Erratic 91A Spike 91A	Erratic 1188 Erratic 1188 Spike 1188 Spike 1188		Seen beginning at t = __, HPOTP shows jitter in both secondary turbine seal drain pressure and temperature. Possible seal anomaly or vibration.

Table 7 Secondary Turbine Seal Cavity Pressure

Feature	Feature	Criteria	Postulate
Erratic 91	Erratic 92		Secondary turbine seal cavity pressure appears erratic or spiking at t = __. Possible seal or sensor anomaly.
Erratic 91	Spike 92		
Spike 91	Spike 92		
Spike 91	Erratic 92		
Peak 91A (t_0 time of peak)	Peak 990 (t_1 time of peak)	t1 > t0; more than cut_width* sigma	HPOTP secondary turbine seal cavity pressure peaks earlier than primary turbine seal drain pressure. Possible seal or sensor anomaly.

Table 8. Seals - Comparing Pressure and Temperature

Feature	Feature	Feature	Postulate
Erratic 951A	Not Erratic 1187	NoSpike 1187	Seen beginning at t = __, HPOTP erratic primary pump seal drain pressure may indicate sensor problem, seal anomaly, or vibration. No effect seen in drain temperature.
Erratic 1187	Not Erratic 951A	No Spike 951A	Seen beginning at t = __, HPOTP erratic primary pump seal drain temperature may indicate sensor problem, seal anomaly, or vibration. No effect seen in drain pressure.
Erratic 951A	Erratic 1187		Seen beginning at t = __, HPOTP shows jitter in both primary pump seal drain pressure and temperature. Possible seal anomaly or vibration.
Spike 951A	Erratic 1187		
Erratic 951A	Spike 1187		
Spike 951A	Spike 1187		

Table 9. Primary Pump Seal

Feature	Feature	Criteria	Postulate
No Flat 951A		at thermal equilibrium	Primary pump seal drain pressure is not flat at t = __. Possible sensor or seal anomaly.
No Flat 1187		at lox seal equilibrium	Primary pump seal drain temperature is not flat at t = __. May be turbine temperature effect, sensor, or seal anomaly.
Diff 951A (current), (previous)		at thermal equilibrium	Primary pump seal drain pressure differs from that of previous test.
Diff 1187, (current), (previous) $temp_1$	Diff 233A, (current), (previous) $temp_2$	1187 at lox seal equilibrium, 233A at thermal equilibrium $temp_1 * temp_2 < 0$	Primary pump seal drain temperature differs from previous test with sign opposite from turbine temperature change.

Table 10. Intermediate Seal Purge Pressure

Feature	Feature	Postulate
Erratic 211	Erratic 212	Intermediate seal purge pressure appears erratic or spiking at t = __. Possible purge system, seal, or sensor anomaly.
Erratic 211	Spike 212	
Spike 211	Spike 212	
Spike 211	Erratic 212	

Table 11. Shutdown

Feature	Postulate
Spike 90 [cut,cut+100.0] Spike 190 [cut,cut+100.0}	HPOTP pump discharge pressure shows spike during shutdown at t = __. Possible momentary shaft hangup.
Erratic 90 [cut,cut+100.0] Erratic 190 [cut,cut+100.0]	HPOTP pump discharge pressure erratic during shutdown. Possible anomaly.
Spike 24 [cut,cut+100.0]	HPOTP turbine discharge pressure shows spike during shutdown at t = __. Possible momentary shaft hangup.
Erratic 24 [cut,cut+100.0]	HPOTP turbine discharge pressure erratic during shutdown. Possible anomaly.

Table 12. Preburner Pump Bistability

Feature	Postulate
Bistability	PBP bistability at thrust level __.

Table 13. Redline Violations

Feature	Postulate
Redline	Redline violated: 'sensor' 'limit type' starting at t = __.

Appendix C - GUI Features

Using a Mouse

The mouse and the pointer which appears on the screen coexist. Moving the mouse causes the pointer to move. Clicking the mouse button activates whichever object lies under the graphical pointer, such as a window or button.

To move the pointer: *move the mouse.*

To click: *press and release the mouse button quickly (usually the far left button).*

To double-click: *press and release a mouse button twice quickly (usually the far left button).*

Selecting a Menu Option

The mouse is also used to select a menu option. Before being activated, a menu appears like a button. Once activated, a list of options appears.

To activate a menu: *move the pointer to the desired menu, then press and hold down the left mouse button.*

To select a menu option: *while the mouse button is still held down, move the pointer to the desired option and then release the mouse button. Or if you wish, you may click and release to display the menu, then place the cursor on the desired option and click again.*

Working With Windows

Once selected (or activated), a window may be moved, resized, shuffled, closed, iconified, and scrolled. Sometimes, there are multiple windows on the screen each providing a view into a different application process.

To activate a window: *move the pointer anywhere on the window and click.*

To move a window: *move the pointer to the upper bar of the window (called the title bar); press and hold down the mouse button; move the mouse to the desired location while still holding down the mouse button; release the button.*

To resize a window: *press and hold down the left or center mouse button the the lower right corner of the window frame; move the mouse outward for a larger window or inward for a smaller window while still holding the mouse button down; release the button when the image of the window (represented by a transparent rectangle) has the desired size.*

To shuffle windows: *only one window can be active when there are multiple windows on the screen. Currently inactive windows appear faded in comparison to the active window. Click on a window other than the active window*

To use a scroll bar: *when more material is available for display than fits in a window, a scroll bar may appear along the edge of a window. Click on the darkened portion of the scroll bar to move up or down one page; move the pointer on top of the slider, press, hold, and move the mouse to scroll incrementally.*

To close a window: *press the mouse button in the upper left corner of the window and hold; move the pointer to the "Close" option and release the button.*

To iconify a window: *click the iconify window button (the left square with a dot in the center in the upper right corner).*

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 1994	Final Contractor Report

4. TITLE AND SUBTITLE

Reusable Rocket Engine Turbopump Health Management System

5. FUNDING NUMBERS

WU–584–03–11
C–NAS3–25882

6. AUTHOR(S)

Pamela Surko

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Science Applications International Corporation
10260 Campus Point Drive
San Diego, California 92121

8. PERFORMING ORGANIZATION REPORT NUMBER

E–9153

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio 44135–3191

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

NASA CR–195388

11. SUPPLEMENTARY NOTES

Project Manager, June Zakrajsek, Space Propulsion Technology Division, NASA Lewis Research Center, organization code 5310, (216) 433–7470.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified - Unlimited
Subject Categories 15 and 20

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

A health monitoring expert system software architecture has been developed to support condition-based health monitoring of rocket engines. Its first application is in the diagnosis decisions relating to the health of the high pressure oxidizer turbopump (HPOTP) of Space Shuttle Main Engine (SSME). The post test diagnostic system runs off-line, using as input the data recorded from hundreds of sensors, each running typically at rates of 25, 50, or .1 Hz. The system is invoked after a test has been completed, and produces an analysis and an organized graphical presentation of the data with important effects highlighted. The overall expert system architecture has been developed and documented so that expert modules analyzing other line replaceable units may easily be added. The architecture emphasizes modularity, reusability, and open system interfaces so that it may be used to analyze other engines as well.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Expert systems; Data reduction; Data processing	90
	16. PRICE CODE: A05

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	