AFHRL-TR-90-79

(2)

AIR FORCE

HUMAN RESOURCES LABORATORY

INTEGRATED MAINTENANCE INFORMATION SYSTEM (IMIS) DIAGNOSTIC MODULE REDESIGN

Garth Cooke
Nicola Maiorana
Theodore Myers

Systems Exploration, Incorporated
5200 Springfield Pike, Suite 312
Dayton, Ohio 45431

LOGISTICS AND HUMAN FACTORS DIVISION
Wright-Patterson Air Force Base, Ohio 45433-6503

December 1990

Final Technical Report for Period August 1989 – September 1990

AIR FORCE SYSTEMS COMMAND
BROOKS AIR FORCE BASE, TEXAS 78235-5601

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the Government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

The Public Affairs Office has reviewed this report, and it is releasable to the National Technical Information Service, where it will be available to the general public, including foreign nationals.

This report has been reviewed and is approved for publication.

JANET MURPHY, 1Lt, USAF
Contract Monitor

BERTRAM W. CREAM, Technical Director
Logistics and Human Factors Division

HAROLD G. JENSEN, Colonel, USAF
Commander

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1990	Final Technical Report – August 1989 – September 1990

4. TITLE AND SUBTITLE
Integrated Maintenance Information System (IMIS) Diagnostic Module Redesign

5. FUNDING NUMBERS
C - F33615-88-0004
PE - 66205F
PR - 1710
TA - 00
WU - 40

6. AUTHOR(S)
Garth Cooke Nicola Maiorana Theodore Myers

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Systems Exploration Incorporated
5200 Springfield Pike, Suite 312
Dayton, Ohio 45431

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAMES(S) AND ADDRESS(ES)
Logistics and Human Factors Division
Air Force Human Resources Laboratory
Wright-Patterson Air Force Base, Ohio 45433-6503

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFHRL-TR-90-79

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This technical report describes Research and Development (R&D) efforts to redesign and enhance the Integrated Maintenance Information System Diagnostic Module (IMIS-DM). The diagnostic module, redesigned in Smalltalk/V and utilizing information from the Content Data Model (CDM), provides technical support to the maintenance technician by furnishing a wide range of capabilities to assist in the selection of an efficient sequence of maintenance tasks.

Object-Oriented (OO) rapid prototyping techniques were used to create a diagnostic module compatible with hierarchical data base concepts employed by the CDM. The enhanced IMIS diagnostic module now provides three modes of assessment (functional, physical, and degraded) whereas the previous version provided only the functional mode. Other IMIS diagnostic module enhancements further expanded diagnostic capabilities with the implementation of algorithms to provide an access group option, "But Not" data entry, presentation of test results, and a revised critically function.

14. SUBJECT TERMS
criticality, diagnostic, fault, feedback analysis, functional assessment, maintenance, physical assessment, rectification, symptom

15. NUMBER OF PAGES
130

16. PRICE CODE

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

SUMMARY

This technical report describes Research and Development (R&D) efforts to redesign and enhance the Integrated Maintenance Information System diagnostic module (IMIS-DM). The diagnostic module is part of an ongoing IMIS R&D effort by the Air Force Human Resources Laboratory (AFHRL) to access and integrate maintenance information from multiple sources and present the information to technicians through a rugged, hand-held computer. The diagnostic module, redesigned in Smalltalk/V and utilizing information from the Content Data Model (CDM), provides technical support to the maintenance technician by furnishing a wide range of capabilities to assist in the selection of an efficient sequence of maintenance tasks.

One of the major outcomes of this effort was a redesigned IMIS diagnostic module. Object-Oriented (OO) rapid prototyping techniques were used to create a diagnostic module compatible with hierarchical data base concepts employed by the AFHRL CDM. The IMIS diagnostic module was reprogrammed using logic-based programming techniques offered by the C Language Production System (CLIPS). In addition, enhanced IMIS-DM assessment methods were developed. The enhanced IMIS diagnostic module now provides three modes of assessment (functional, physical, and degraded) whereas the previous version provided only the functional mode. Other IMIS diagnostic module enhancements further expand diagnostic capabilities with the implementation of algorithms to provide an access group option, "But Not" data entry, presentation of test results, and a revised criticality function.

Enhancements to the IMIS diagnostic maintenance environment give maintenance technicians a test result data validity check and the ability to specify unsuccessful maintenance actions. All enhancements, with the exception of those to the physical assessment module, were implemented in the OO rapid prototyping environment offered by Smalltalk/V. The physical assessment module was implemented in Smalltalk/V, logic-based Programming in Logic (PROLOG).

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

PREFACE

This report documents a Research and Development (R&D) effort to redesign, enhance, and improve the Integrated Maintenance Information System diagnostic module (IMIS-DM) and the environment for which this module functions. The R&D efforts were performed for the Air Force Human Resources Laboratory (AFHRL), Combat Logistics Branch (LRC), under the terms of contract #F33615-88-C-0004, Task Order #0012. The Task Monitor was Lt Janet Murphy, AFHRL/LRC.

Research was performed by the Dayton regional office of Systems Exploration, Inc. (SEI). Principal investigators were Garth Cooke, Nicola Maiorana, Theodore Myers, and Johnnie Jernigan.

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

Purpose

The Air Force Human Resources Laboratory (AFHRL) is engaged in a long-term program to improve information presentation in the maintenance environment. Research and Development (R&D) has led to an Integrated Maintenance Information System diagnostic module (IMIS-DM) capable of utilizing existing data parameters and producing effective isolation and repair recommendations. This report describes the work performed to design the IMIS diagnostic module in Object-Oriented (OO) Smalltalk/V and logic-based "C" Language Production System (CLIPS), to implement enhancements using the rapid prototyping techniques and logic-based programming of Smalltalk/V, and to define feedback analysis parameters and functions pertinent to the maintenance diagnostic and flightline community.

Background

The IMIS diagnostic module[1], or Maintenance Diagnostic Aiding System (MDAS), was designed to help aircraft maintenance technicians identify and repair a malfunctioning weapon system. The diagnostic module's key feature is that its algorithms are designed to minimize time to repair an aircraft rather than time to isolate a fault. This philosophy incorporates rectification actions into the overall diagnostic sequence when appropriate, resulting in the repair of faulty components and physical damage in minimum time. The diagnostic module employs special subroutines modifying the split-half dependency model by applying fault/symptom/component matching, component histories, probabilistic data, logistics constraints, and operational constraints.

Fault/symptom matching is employed throughout the diagnostics modeling scheme. The term "fault" is used to describe a functional or physical manifestation of some low-level physical failure. Under normal circumstances, it is likely there would be only one fault present in a system at any time. The purpose of a diagnostics task is to isolate that fault and rectify conditions causing it. Throughout this report, we use the term fault to refer both to the actual fault(s) present in the system (that which is bad) and to all possible faults that can cause a symptom. A symptom is a machine-generated code or a verbal description indicating a malfunction exists within a system. The symptom implicates one or more possible faults. Use of these terms can be confusing in a hierarchically arranged data base such as the Content Data Model (CDM) (AFHRL, 1989), because what is called a fault at one level of the hierarchy may be referred to as a symptom in a lower level of the hierarchy. This relationship is illustrated in Table 1.

In the radar system example shown in Table 1, the lists of possible faults are neither all inclusive nor necessarily correctly stated. The lists are merely illustrative. As each fault is isolated at any level of the hierarchy, that fault can become a descriptive symptom of faults lower in the hierarchy.

Previous R&D efforts in diagnostics produced an almost purely functional assessment module for isolating and repairing faulty components. Therefore, the software once called MDAS is now designated as the functional assessment module. Below is a brief discussion of the functional assessment module's initialization and operations processes. Detailed information about algorithms and operations can be found in Integrated Maintenance Information System (IMIS) Diagnostic Module (Cooke, Jernigan, Maiorana, & Myers, 1990).

---

[1]The IMIS-DM is based upon the Maintenance Diagnostic Aiding System (MDAS) which was developed in an earlier research effort.

Table 1. Hierarchical Fault/Symptom Logic

Radar System Example		
Maintenance level	Symptom	Faults
On-Equipment	Radar Inoperative	Transmitter Bad Receiver Bad Antenna Bad[a]
Off-Equipment	Antenna Bad	Motor Frozen Servo Inoperative Power Circuit Bad[a]
Off-Equipment	Power Circuit Bad	Transformer Bad Resistor Bad Capacitor Bad

[a] Fault isolated by testing (becomes symptom at next maintenance level).

During initialization, data can be entered in the functional assessment module automatically and manually. Automatic data collection loads system-specific data files from the CDM data base and allows downloading of system health information from an aircraft data bus. The operator performs manual data entries such as symptoms, parts and test equipment availability, critical states, and aircraft configuration. Figure 1 shows the sequencing of algorithms and analyses performed by the initialization process and functional assessment module.

After initialization, the functional assessment module uses automatic and manual data input to evaluate fault combinations and rank tests. The module then compares tests--by time analyses and failure probabilities--as to repair or replace activities. That is, it identifies the action with the highest likelihood of fixing the problem in the least amount of time. Three lists of ranked tests and/or rectifications can be selected and presented to the maintenance technician: (a) ranked tests, (b) ranked rectifications, and (c) interleaved tests/rectifications. Although a "best" action is recommended, the technician has the option of choosing any of the listed options. When the technician selects a test or rectification/maintenance action, the presentation system displays technical order instructions for performing the selected activity. If the selected action is a test, the functional assessment module evaluates the test outcome and available options to determine the status of the faults. If the selected action is a rectification or maintenance action, the technician is instructed to perform a functional check. Procedures are provided, a functional check is performed, and fault/symptom status is reinitialized as required. This process continues until the fault is isolated and the system repaired.

II. IMIS DIAGNOSTIC MODULE (IMIS-DM) DESIGN

Previous versions of the IMIS diagnostic module provided isolation and repair decisions from a purely functional assessment standpoint (i.e., symptom occurrence) and did not provide a means to rectify faults when the list of suspected faults was exhausted (degraded assessment) or when faults were the result of some form of physical manifestation (physical assessment). Degraded and physical assessment modules were developed to eliminate these weaknesses. Thus, the IMIS-DM is composed of three submodules: (a) functional, (b) degraded, and (c) physical. As

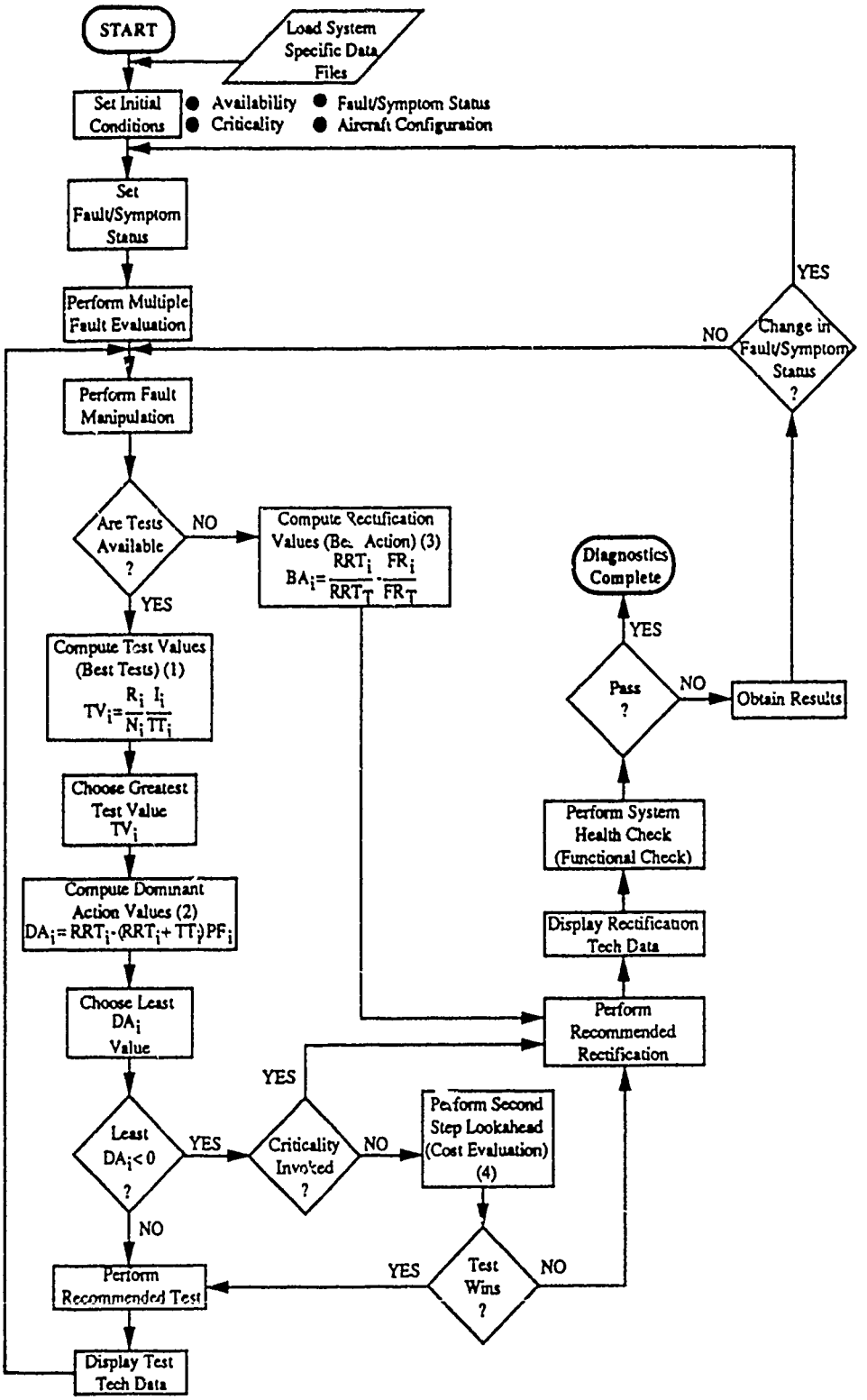


Figure 1. Logic Flow.

a result of the development of the degraded and physical assessment modules, some means of control over the activation and processing of the three modules had to be developed, as well as a means to interface diagnostic information with the presentation system and the CDM data base. The development of the IMIS controller and the diagnostic module controller provides the means to extract diagnostic information from the hierarchical CDM data base, to deliver it to the presentation system, and to control the operations of the assessment mode selection.

Four IMIS diagnostic module developments are described in detail in this section: (a) the OO Smalltalk/V module development, (b) the OO PROgramming in LOGic (PROLOG) module development of the physical assessment module, (c) the degraded mode module development, and (d) the CLIPS IMIS diagnostic module development. The OO Smalltalk/V module development included (a) the reprogramming of the original MDAS, now called the functional assessment module, and (b) the incorporation of recommended enhancements as described in Section III. The physical assessment module was implemented with Smalltalk/V PROLOG. The IMIS diagnostic functional module was also redesigned into a purely logic-based environment offered by CLIPS, a language developed by the National Aeronautics and Space Administration (NASA).

CDM Compatibility and Hierarchical Modeling

The IMIS diagnostic module was designed to operate in a Technical Order (TO) environment similar to that currently used for TOs. Current concepts for electronic presentation of technical data were not included in that version. The earlier data base version was used to store and retrieve information for diagnostic module and presentation system operations, and employed relational data base modeling schemes. The newly developed CDM employs hierarchical data modeling techniques, thus providing a more efficient data system by eliminating duplication of effort, and is more effective in data access and format. Hierarchical data modeling models system data in stages or levels, providing the mapping of systems and corresponding subsystem classes. For example, in Figure 2, the highest system in any aircraft data base would be the aircraft itself; divisions of the aircraft system (e.g., Fire Control System, Propulsion System) would be mapped to the aircraft system. In turn, these lower level aircraft subsystems could then be divided into smaller systems or subsystems (e.g., Radar, Mission Control System) and mapped to the respective aircraft subsystem. This process of system/subsystem mapping is hierarchical data modeling within the CDM.

Previous versions of the IMIS diagnostic module were intended as a maintenance aid to be used at the organizational maintenance level, operating on faults modeled at the Shop Replaceable Unit (SRU) to isolate faults to an implicated Line Replaceable Unit (LRU). This design placed previous versions of the IMIS diagnostic module in the middle of the CDM's hierarchical structure. The new OO IMIS diagnostic module is designed to be compatible with all CDM hierarchical data modeling levels; and it extracts, computes, and stores diagnostic information using the diagnostic controller and the IMIS controller.

IMIS Controller

The IMIS controller is an executive system that controls and manipulates three subsystems: (a) an applications system (such as IMIS-DM, pre/post flight, phase inspection, and weapons load), (b) the data base module, and (c) the presentation system. Figure 3 illustrates the system. The applications system is identified as the diagnostic module.

The diagnostic module is comprised of four major submodules: (a) the diagnostic controller, (b) the physical assessment module, (c) the functional assessment module, and (d) the degraded mode module. The diagnostic controller module regulates data items, diagnostic

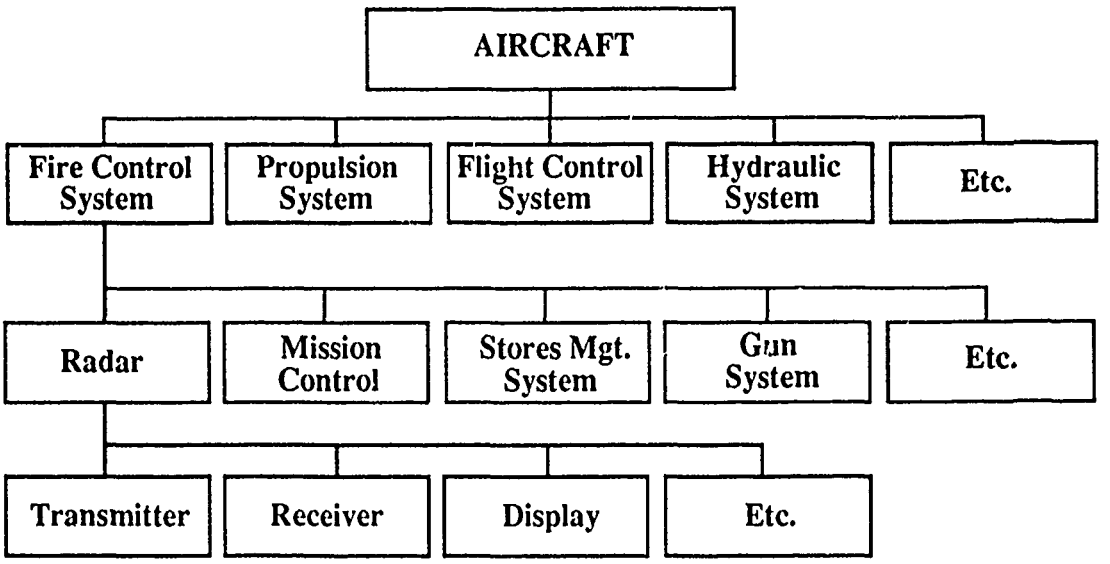


Figure 2. CDM Hierarchical Modeling.

groupings, performance of its submodules, and interfacing to the IMIS controller. Interfaces between the IMIS controller and the diagnostic controller provide the means to extract diagnostic data from the data base and present information to the presentation system. The functional assessment module was described briefly in the background; the remaining diagnostic modules are new developments and are described in this section.

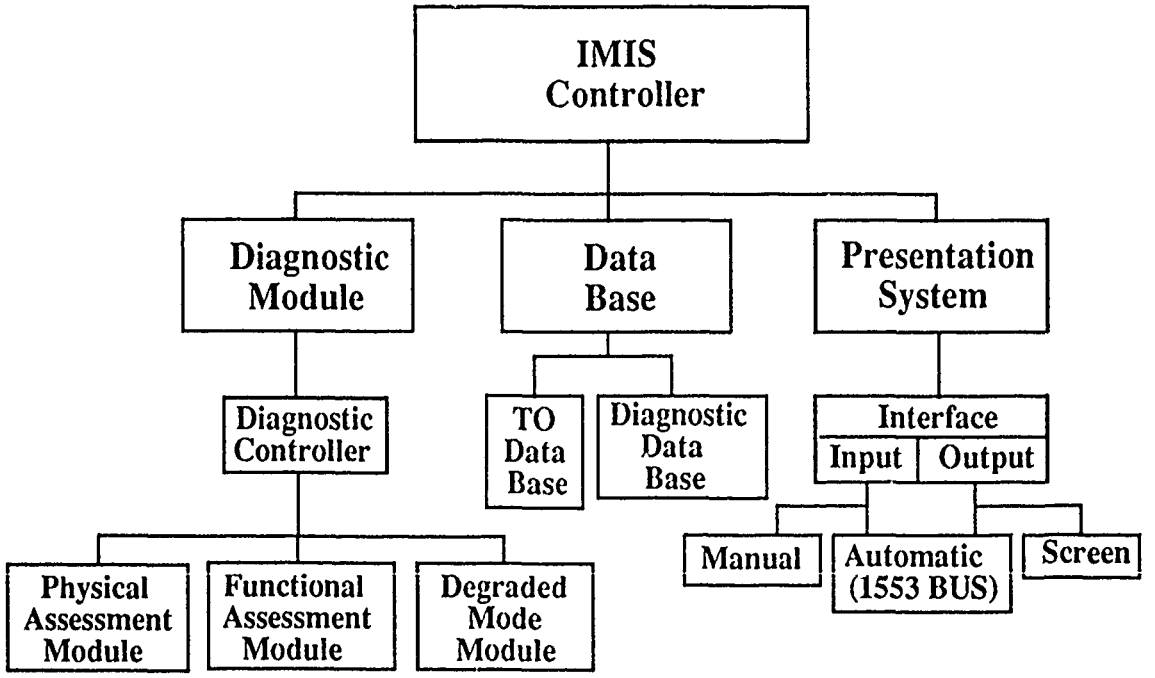


Figure 3. IMIS Controller System.

The data base module, in CDM hierarchical format, contains both TO and diagnostic data information. The TO information consists of procedural text, graphical illustrations, and data element information; the diagnostic data consist of data elements and graphic functional descriptors. Diagnostic data are used mainly by the IMIS diagnostic module; the TO information is used mainly by the presentation system.

The presentation system provides interfaces for aircraft system health checks, maintenance technician input, display of diagnostic module results, maintenance tasks, and system-specific graphics. System health checks from Military Standard 1553 data bus downloads and maintenance technician input give the diagnostic module pertinent information about the aircraft under investigation, logistics constraints, operational constraints, and task performance. The diagnostic module gives the presentation system information for fault isolation and repair recommendations. The data base module, in turn, supports both the diagnostic module and the presentation system.

Smalltalk/V Module Development

The original diagnostic module was written using the C programming language. We converted this module to Smalltalk/V. There are two reasons why Smalltalk was selected as a development language. The first is the rapid prototyping capabilities available through Smalltalk and facilitated through the Smalltalk environment. This environment contains most of the low-level functions used in software development. The environment also allows a programmer to compile and test smaller pieces of code within the environment. Moreover, the Smalltalk environment allows us to reuse the code. The second reason for selecting Smalltalk is for ease of integration into the presentation system, which is also developed in Smalltalk.

Smalltalk is a high-level, OO programming environment. The basic building blocks in the Smalltalk environment are objects. Objects can be as simple as numbers, strings, and arrays; or they can be as complex as menus, data base managers, and diagnostic modules. Everything is an object in Smalltalk. Objects can store data in variables, perform functions, or both. All objects are types or instances of some Smalltalk class. An instance is an object described by a particular class type. A Smalltalk class describes the variable names and the functions for instances of itself. The variable names, also known as instance variables, are descriptive tokens where objects store data. The data stored in the instance variables are other objects. The class also describes the methods for its instances. Methods have the names of actions or functions that objects can perform. The method description describes each method's function in terms of Smalltalk code. The method names are passed to the object as messages to perform a particular action.

The Smalltalk/V variant of the Smalltalk language was chosen over Smalltalk-80 for two reasons. One, Smalltalk/V required much less overhead than Smalltalk-80. Secondly, the technology existed to port Smalltalk/V to the portable computer that will be used in future IMIS demonstrations.

Diagnostic Module's Smalltalk Class Structure

To implement the Smalltalk version of the diagnostic module, several classes were created: controller, previous action, diagnostics, sets, a combination generator, combination, tasks and ranked task. The diagnostics and controller classes are the primary classes used in the module redesign and are discussed below.

1. The diagnostics class consists of the instance variables and functions needed to perform diagnostics. Each instance of this class is equivalent in functionally to the diagnostic

module written in C. The difference is that the C version allowed only one diagnostic module; now there can be many.

2. One instance of the controller class is needed to perform diagnostics. This instance creates, monitors, and controls one or more diagnostic groups. This process is described in Section II. Furthermore, the interface into the diagnostic module is defined by the controller class. This feature gives the presentation system a focal point of communication to the diagnostic module and the ability to port the diagnostic module to different presentation systems. In the C version, this interface was spread throughout the presentation system, and porting the diagnostic module was more difficult. Details on the OO Smalltalk/V module development are in the appendices.

Diagnostic Module Controller

The diagnostic controller controls the creation of diagnostic groups. Diagnostic groups are independent diagnostic problems. To determine the independence of problems, the observed symptoms are evaluated. Observed symptoms possessing common tasks (tests or rectifications) are related to each other and reside in the same diagnostic group. The implicated faults from a symptom are used to determine the tasks of a symptom. The implicated faults make up fault trees by having subfaults. The bottom of any branch of the tree is determined by the faults that have a rectification task. All the faults at the bottom of the tree are used to determine which tasks relate to a given symptom. This is illustrated in Figure 4. In the figure, Symptom 1 is linked to Tasks 1, 2, and 3.

The processing is started when observed symptoms are passed to the diagnostic controller. Each symptom is evaluated for tasks common to any existing diagnostic group. If a diagnostic group has tasks in common with the symptom, the symptom is added to it. When the current list of diagnostic groups is exhausted and no diagnostic groups are found in common with the symptom, a new diagnostic group is created.

This process is illustrated using Figure 5 as an example symptom set. In the figure, Symptom 1 is independent of Symptoms 2 and 3 because there are no common tasks. Symptoms 2 and 3 are interdependent because they both contain links to Task 5. The process explained above would create two diagnostic groups. First Symptom 1 is evaluated, and, since no diagnostic groups exist yet, the first one is created. The second diagnostic group is created when Symptom 2 is evaluated because Symptom 2 does not have any tasks in common with any of the current diagnostic groups. When Symptom 3 is evaluated, it is found to have commonality with the second diagnostic group. Therefore, it is added to the second diagnostic group.

Physical Associations Model Development

The diagnostic module, originally designed to evaluate fault isolation and repair alternatives from almost a purely functional standpoint, has been enhanced to perform both functional and physical assessment. When diagnostics are approached from a purely functional standpoint, we cannot adequately address events causing malfunctions of other components, or malfunctions caused by a nearby physical event. For example, a technician may enter a compartment of an aircraft and observe that hydraulic fluid has leaked all over the bay, causing a failure in an LRU. Repair of that LRU would not be appropriate until the hydraulic line is repaired and the bay is cleaned. Many external causes of the functional problem (i.e., aircraft battle damage, bird strikes, environment extremes) could create problems with the system under investigation. Hence, some physical model is needed to provide an efficient cause and effect or physical association isolation and repair strategy.

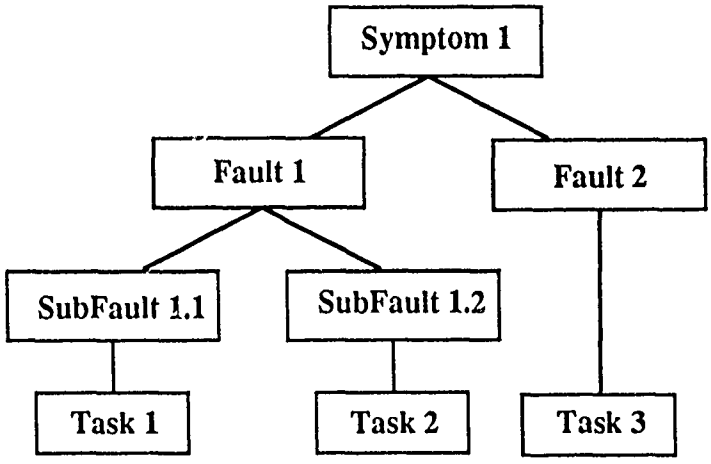


Figure 4. Fault Tree Illustration.

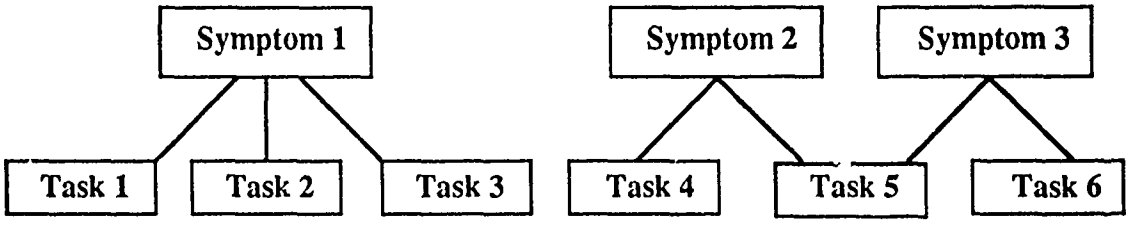


Figure 5. Symptoms with Common Tasks.

In order to work with physical associations, we need to look at what makes a physical association reasonable. One key element is physical proximity. However, proximity alone is not enough. There must be a physical event occurring that can affect systems, components, or parts near the event for a valid physical association. A physical event in this context implies that some foreign agent can act externally to the affected component and cause a failure. This implication, then, implies that there must be a source of the foreign agent and that the component(s) in the vicinity must be vulnerable to damage by that agent.

If we consider only the battle damage source, there are many assessment models available that may prove more effective for this limited role than this modified diagnostic module. However, if we consider other sources, we need to look inside the weapon system to find potential sources of the damaging physical agents. When we look inside the weapon system for these sources, we define the limits of the universe of possible physical associations. Most normal physical hazards associated with operating in an airborne environment are already built into the Mean Time Between Failure (MTBF) and the fault weightings considered in the functional diagnostic module (e.g., routine g-loads, normal vibration levels, operating temperature extremes, humidity, and so on). Consequently, the physical associations model must address hazards outside the range of "normal" hazards associated with operating in an airborne environment.

Therefore, the hazards that must be modeled by the physical module are few. Among these are:

1. temperature extremes, specifically high temperatures;

2. liquids such as fuel, lubricants, hydraulic fluid, and water; and

3. physical abuse. This is the most widespread category because it includes both internal and external sources and has a wide range of potentially severe effects. These sources can be internal (explosion of Cartridge Activated Device (CAD); rupture of pressure vessels; slow burning/misfire of CAD; and loss of containment of high-energy, spinning devices) or external (dropped objects, bird strike, mid-air collision, Foreign Object Damage (FOD), and battle damage).

Physical Effects Mapping

With a fully developed restricted hazards list, we must identify, within some boundary (e.g., an avionics bay, an engine bay), each of the components containing hazards either to itself or to other components within the boundary. Finally, we must identify those components within the boundary that are vulnerable to these hazards. Vulnerable, in this case, refers to the functional model and implies some component may not operate within prescribed functional limits because of the effect created by a hazard normally contained within some other component.

Assuming that data to support the above discussion are available, then the diagnostic model must be altered to consider the effects of these physical relationships. If, during a diagnostic or other maintenance task, evidence of the presence of a physical hazard is discovered, the maintenance technician is faced with two problems. First, he must identify the source of the hazard and rectify the failure that produced the hazard.[2] Then, he must identify and, if necessary, repair any components which have been affected by the hazard. The data to support this scheme could be represented as in Table 2.

Table 2. Hazard Source and Effects Mapping

Rectification (Rect)	Hazard (H)	Vulnerability (V)	Source Fault(s) F(s)	Effect Fault(s) F(e)	Location (LOC)
A	a	-	1	-	BAY 1
	b	-	2	-	
	-	c	-	1,3,4	
B	c	-	5	-	BAY 2
	-	a	-	6,7,8	
C	-	c,a	-	9	BAY 1
Rect ID	Hazard contained in Rect	Hazards Rect is vulnerable to	Faults which can lead to Hazard release	Functional faults which may result from exposure to Hazard	Location for Rect

[2]The individual technician is central to the discussions in this report. For simplicity, we have used the singular pronoun "he" to designate the individual technician (whether that person is a man or a woman).

Physical Model Operation

The logic flow for physical associations modeling appears as in Figures 6-7. This logic flow maximizes the capabilities built into the current functional assessment module and expands upon the processing and modeling schema. The branching and control mechanisms have been built into the current diagnostic module using the logic-based PROLOG of Smalltalk/V.

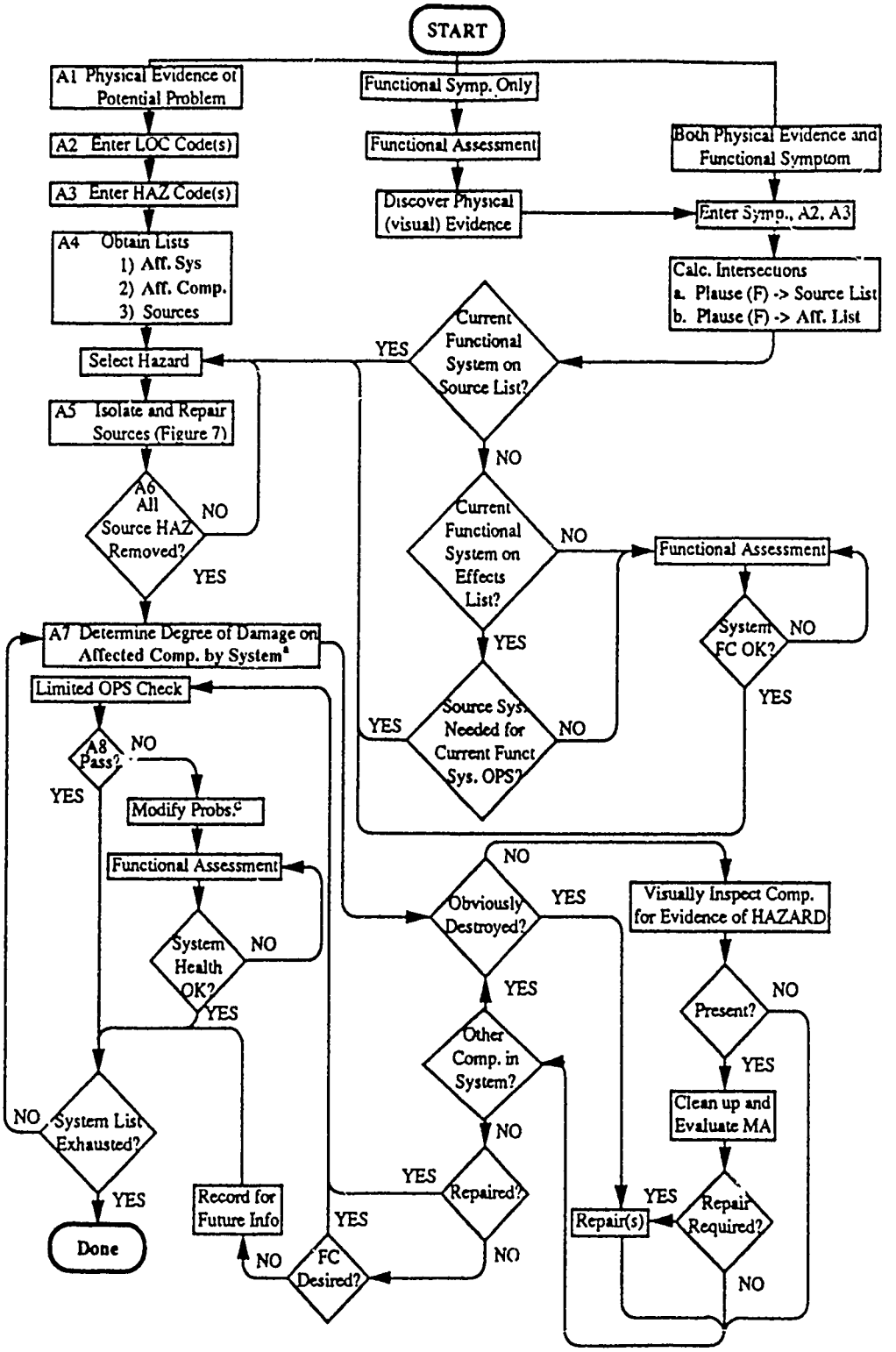
Upon initialization, manual and automatic system health information is entered from within the functional assessment module. If physical evidence of a hazard is encountered, the technician can select the physical assessment mode and proceed down one of the two paths of the physical assessment module described below. The diagnostic controller module determines which path to proceed with, based on the data entries and diagnostic groupings.

1. If the physical assessment mode is selected and no functional symptoms are present, the technician is prompted with menus to select an access group and the hazards observed within that access group. Current implementation of the physical module utilizes access groups for location, but future developments should include both access group and three-dimensional mapping to identify physical locations. The technician can select as many access groups and hazards associated with each access group as he wishes. Once manual access groups and hazards are chosen, the physical assessment module determines and produces lists of affected systems, affected components, and sources of the hazards from a search of elements within the CDM data base. If there is more than one source of the observed hazard, the technician is prompted for the hazard which he wishes to attack first. The physical assessment module then proceeds with isolation and repair of the source hazard.

2. The second mode of operation of the physical assessment occurs when symptoms are present, evidence of a hazard is observed, and the technician elects to select the physical assessment mode of operation. The diagnostic controller module redirects efforts to physical isolation and repair assessment without losing the information gained from previous actions performed in the functional assessment. The physical assessment module first prompts the technician for access groups and hazards and obtains lists as if the technician were proceeding down the first path. After the technician has completed his entries, the same lists as explained above are produced, the physical assessment module obtains lists of intersecting faults and source components, and intersecting faults and affected components. From this point, the physical assessment module proceeds as follows:

a. If the system under investigation within the functional assessment module appears within a list of source systems, the user is prompted for a hazard selection if more than one exists. The physical assessment module then proceeds to isolate and repair source components of the selected hazard by system.

b. If the system under investigation within the functional assessment module is not on the source system list, the affected systems list is checked for that system's presence. If the system is not present in the physical module's affected system list, the functional assessment is reinstated to alleviate the system upon which it had been working. Upon successful functional system repair, the technician can reenter the physical mode of operation and continue to select a hazard, and isolate and repair sources of the hazard observed by system.



s = systems
c = components
Plause(F) = The set of plausible functional faults

Figure 6. Physical Assessment Mode.

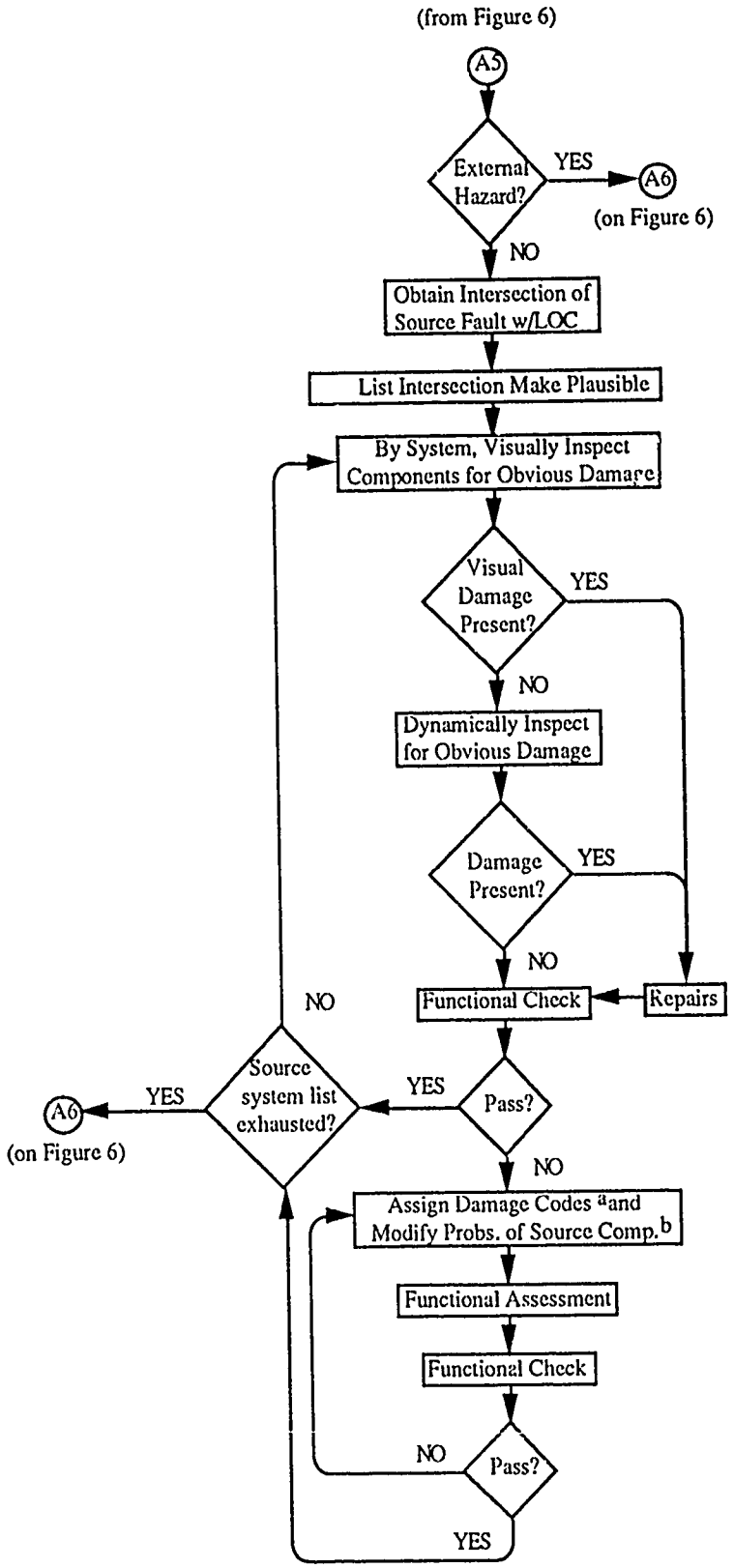


Figure 7. Isolate and Repair Sources.

Note.

[a]A description of the damage codes and formulas for modifying probabilities of components and faults are as follows:

Damage Code - (DC)

Code	Value	Desc
DS	100	Destroyed
DM	80	Damaged (dented, soaked, scorched)
SP	20	Suspected (dinged, scratched, dampened, liquid spots)
OK	1	No Apparent Effect

[b]Modifying probabilities of source components and faults. The following formulas are evaluated to obtain normalized modified source component and fault probabilities for a given location. Although Formulas 3 and 4 are listed only once, both component and fault probabilities are calculated separately using the same formulas.

$$Prob_{i,1} = \frac{\sum^{Comp} \frac{1}{MTBF\,(Fs)}}{\sum^{LOC} \frac{1}{MTBF\,(Fs)}} \quad (1)$$

$$Prob_{i,1} = \frac{\frac{1}{MTBF(Fs)}}{\sum^{LOC} \frac{1}{MTBF(Fs)}} \quad (2)$$

$$Prob_{i,2} = Prob_{i,1} * DC \quad (3)$$

$$Prob_{i,F} = \frac{Prob_{i,2}}{\sum Prob_{i,2}} \quad (4)$$

Where, DC = component damage code value,
Fs = source fault,
Comp. = component,
LOC = location,
$Prob_i,1$ = probability of the ith source component or fault,
$Prob_i,2$ = modified probability of the ith source component or fault, and
$Prob_i,F$ = normalized modified probability of the ith source component or fault.

[c]Modifying probabilities of affected components and faults. The following formulas are evaluated to obtain normalized modified affected component and fault probabilities for a given location. Although Formulas 7 and 8 are listed only once, both component and fault probabilities are calculated separately using the same formulas.

$$Prob_{i,1} = \frac{\sum_{1}^{Comp} \frac{1}{MTBF(Fe)}}{\sum_{1}^{LOC} \frac{1}{MTBF(Fe)}} \quad (5)$$

$$Prob_{i,1} = \frac{\frac{1}{MTBF(Fe)}}{\sum_{1}^{LOC} \frac{1}{MTBF(Fe)}} \quad (6)$$

$$Prob_{i,2} = Prob_{i,1} * DC \quad (7)$$

$$Prob_{i,F} = \frac{Prob_{i,2}}{\sum Prob_{i,2}} \quad (8)$$

Where, Fe = affected fault,
$Prob_{i,1}$ = probability of the ith affected component or fault,
$Prob_{i,2}$ = modified probability of the ith affected component or fault, and
$Prob_{i,F}$ = normalized modified probability of the ith affected component or fault.

---

c. If the functional system was on the affected system list in (b) above, the technician is prompted with a menu to determine if the source system is needed for current functional system operations check. If so, the technician is prompted with a list of hazards. If more than one hazard exists, isolation and repair of sources proceed. If the source system is not needed for current functional system operations checks, the functional module is reinstated and diagnostics continue as in (b) above.

When the isolation and repair of sources function is entered, the first action performed determines if any sources exist. Figure 7 provides the logic flow for alleviating source faults (A5 on Figure 6). If there are no internal sources of the named hazard (e.g., bird strike, bullet hole in avionics bay), the physical assessment module proceeds to isolate and repair affected components. When sources of hazards are present, the physical assessment module attacks the selected hazard by producing a list of plausible source components within the location by system. The plausible components of a system are presented to the technician for visual damage inspection; repairs can also be selected. If the technician has not made any repairs during the visual inspection, the same components are presented for dynamic inspection during system "power up," and repairs can be selected. When repairs have been completed during the visual or dynamic inspection process, the technician is prompted for a functional check of the system previously investigated. If a pass is exhibited, the physical assessment module either continues to isolate and repair sources within other systems, if there are any present, or, if not, continues to isolate and repair other source hazards in the same manner. If a fail is exhibited from the functional check, the physical assessment module prompts the technician to assign damage codes to the components under investigation and modifies the probabilities of the corresponding rectifications and intersecting functional faults. (Damage codes and formulas are described below.) The modified plausible fault probabilities and rectification probabilities are then passed to the functional assessment module for further isolation and repair, until a pass is exhibited on the functional check. When a pass is

exhibited by a functional check of that system, the physical module either continues to isolate and repair sources within other systems, if there are any present, or, if not, continues to isolate and repair other source hazards in the same manner.

When all source faults are removed from consideration, the physical assessment module begins to isolate and repair affected components. The approach to physical assessment of affected components is to attack affected components by system, prompting the technician to assign damage codes and repair obviously destroyed components first. When all obviously destroyed system components have been repaired, damaged system components are investigated and Maintenance Actions (MAs) and repair menus are presented to the technician for selection. When all repairs and MAs have been performed, the physical assessment module determines whether any repairs had been made. If no repairs have been performed, the technician is asked if he would like to perform a functional check. If the technician does not wish to perform a functional check, the physical assessment module continues with the next system in the same manner; or if all systems have been exhausted, diagnostics end. If the technician wishes to perform a functional check or if repairs were made to affected components within the system, a limited operations (OPS) check is performed. If the outcome of the limited OPS check is a pass, the physical assessment module continues with the next system in the same manner; or if all systems have been exhausted, diagnostics end. If the limited OPS check returns a fail, the intersecting faults from the OPS check and the rectification probabilities of the affected system are modified and passed to the functional assessment module for further isolation and repair. When repairs on the system are completed using the functional assessment module and verified by a system health check, the isolation and repair can be activated and proceed until the system list is exhausted.

As shown, the physical assessment module's logic flow maximizes the maintenance technician's abilities to evaluate and act upon physical evidence without being delayed by the details of the general technical data required to identify, clean up, and evaluate hazard exposure. However, the processing is available to provide additional information to assist the novice through the details if necessary.

OO PROLOG Physical Associations Module

Although Smalltalk/V offers well-developed rapid prototyping techniques within an OO environment, it also offers the opportunity for logic-based programming or PROLOG. The Smalltalk PROLOG language, developed from Smalltalk OO code, provides a logic-based environment and functions to develop expert systems. With the development of the physical assessment module theories and operations described above in this section, an expert or logic-based system could more closely duplicate this logical decision process designed from a maintenance technician's viewpoint.

The first step in implementing the logic-based physical assessment module was to identify and develop a mock hierarchical CDM data base containing all of the necessary data elements required of the physical module. This was simply an extension of the newly developed functional model's mock CDM data base and was implemented within the Smalltalk OO environment. Initial format and mapping of the physical module's CDM elements are described below in Table 3. Note: These mock CDM elements are for testing the functionality of the physical associations module; ongoing investigations are being performed to incorporate the information into the actual CDM.

Table 3. CDM Requirements for Physical Associations

Rect Hazard ID (CDMrectHazard) Rectifications (CDMrect) Locations (CDMaccessGroup) vulnerableHazards Hazards (CDMhazards) containedHazards Hazards (CDMhazards) System (CDMsystems)	Hazards (CDMhazards) Hazard Codes (code) Hazard Meaning (meaning) Damage Codes (CDMdamageCodes) Damage Codes (code) Damage Meaning (meaning) Systems (CDMsystems) System Code (code) System Meaning (meaning)

The initial test data base for use by the physical module contained rectification hazard IDs, or CDMrectHazards. Each CDMrectHazard consists of the following elements:

1. A rectification pointer (CDMrect) containing mapped fault pointers, and a location. The locations currently used for physical module testing are access groups.
2. A vulnerable hazards (vulnerableHazards) list of pointers to physical hazards (CDMhazards) to which the rectification and associated component are vulnerable.
3. A contained hazards (containedHazards) list of pointers to physical hazards (CDMhazards) which the rectification and associated component are capable of releasing as a result of the component failure.
4. A system (CDMsystems) is a pointer to a system of which the rectification (and associated component) is a member.

The hazards element (CDMhazards) is a list of the possible hazard codes with pointers to their meaning. For instance, the hazard code of HYDFLD has a hazard meaning of hydraulic fluid, and the hazard code HTEMP has a hazard meaning of high temperature. Also, with each rect hazard (CDMrectHazard), there is a pointer to a system. The system (CDMsystems) is a pointer to a particular system, which contains the rectification and a pointer to the associated component. It is identified by a code (systemCode) and meaning (systemMeaning). The code HYD would have a meaning of hydraulic system, and the code AVN would mean avionics. The last element installed into the mock data base contains damage codes (CDMdamageCodes). The CDMdamageCodes are a list of codes and meanings. The meaning is a description of the damage code incurred, and the code contains a value of the damage. For example, the damage meaning of "destroyed" would have a corresponding code with a value of 100. The damage code values and meanings were explained in Figure 6.

After implementing the above CDM elements, we identified the constraints of the Smalltalk/V PROLOG environment. The major constraint imposed was that Smalltalk/V PROLOG had no input or output functions; thus, all user and data interfaces had to be developed within the Smalltalk/V OO environment to test the accuracy of the physical module. The OO Smalltalk/V methods were therefore developed to help interface with the functioning and testing of the logic-based physical assessment module.

The functional diagnostics controller was used to simulate the CDM data capture and manipulation of the IMIS controller, and the user interface of a presentation system. Two classes were created to implement and integrate the Smalltalk PROLOG version of the physical assessment

module into the overall IMIS-DM. These two classes, PhysDiagnostics and the PhysModel, are discussed below.

1. The PhysModel class consists of the rules and functions needed to perform the logical diagnostic decision process of the physical assessment module. This class is a subclass of the PROLOG class, which is a subclass of Logic. Once lists of pertinent data are passed from the MDAScontroller class to the PROLOG data base, rule firing dictates the logical approach of physical diagnostics as described in the above section. If any information or user interface is required, the PhysModel simply sends a response to the OO Smalltalk/V IMIS controller to access and perform menuing and data transfer for technician input. Responses are then passed to the PROLOG dictionary. When the functional assessment module is needed to continue diagnostics, the PhysDiagnostics module is activated; it retrieves and manipulates the test and rectification information from the PhysModel.

2. The PhysDiagnostics class is comprised of the instance variables and functions needed to perform functional assessment and interfacing with the PhysModel of Smalltalk/V PROLOG. The main objectives of this class are to act as an interface between the Smalltalk/V MDAScontroller and the PROLOG PhysModel, and to perform assessment of suspected faults returned from the PhysModel and functional checks.

Degraded Mode

The degraded mode can occur when the diagnostic module can no longer recommend an action because all suspected faults--given symptom occurrence--have been eliminated from consideration. This situation can occur if the diagnostic module is given incorrect or incomplete data. At this point, normal diagnostics can be suspended if the diagnostic module is placed in the degraded mode by the technician.

The objective of the degraded mode is to find a test that fails and results in a plausible set of faults, and/or to perform a rectification that passes a system health check. In certain situations, the technician may choose to put the system into degraded mode in one of two ways: (a) He decides the diagnostic module is no longer helping in troubleshooting, or (b) the diagnostic module recommends reverting to the degraded mode.

The diagnostic module will recommend degraded mode of operation if:

1. a symptom is present but all suspected faults that could have caused the symptom are eliminated from consideration either by passed tests or by the correction of another symptom; or,

2. rectifications have been performed for the second time on the same components where the fault is suspected; or,

3. a symptom is confirmed present but is not in the data base.

Although the maintenance technician has full control of degraded mode selection, degraded mode is necessary to continue diagnostics when:

1. the diagnostic module recommends the degraded mode based on the above occurrences; or,

2. the technician does not agree with any of the diagnostic module's action choices; or,

3. Can Not Duplicate (CND)/Intermittent (unverifiable starting points) messages are received; or,

4. the technician chooses.

When degraded mode is selected, the physical and functional assessment modules are suspended by the diagnostic controller module and a message appears notifying the technician that has entered degraded mode. This message remains on the screen at all times during degraded mode assessment.

While in degraded mode, the physical and functional assessment modules are not able to provide the technician with any recommended actions. To aid the technician, a smart Table Of Contents (TOC) is created. The smart TOC consists of two lists. The first list contains ranked rectifications based on component MTBFs. The components with the lowest MTBFs are ranked highest on the list. The second list contains an ordered list of tests based on probability of failure, calculated by summing the failure rates of all spanned faults. Moreover, to provide a more accurate test ranking, any information gained during physical and functional assessment is used to alter the test's failure probabilities. When modifying the test's failure probabilities, exculpated faults are not used in computations of tests containing them.

Upon display of the TOC, the technician selects a test or rectification from the ranked lists and performs the selected action by following the procedures for that action. Depending on the action performed and its results, diagnostics can proceed in several ways. At the completion of any action, the maintenance technician must either suspend diagnostics, select another action from the TOC, or exit the degraded mode. The technician must consider the precedences described below, which reflect a logical continuation of isolation and repair.

1. If a test is selected during degraded mode assessment and a fail result is exhibited, a new plausible set is established. The degraded module then records the new plausible set of faults for further isolation and repair. Once a new plausible set of faults is established, degraded mode can be exited and physical or functional assessment can proceed from the new plausible set.

2. If a test is selected during degraded mode assessment and a pass result is exhibited, the degraded module records the exculpated faults, eliminates the performed test from consideration, and reranks the TOC's test list. The maintenance technician can then select and perform another action from the TOC.

3. If a rectification is selected and performed during degraded mode assessment and changes to the system are exhibited as a result of a functional check following the rectification, the degraded module records the appearance of new symptoms and/or existing symptoms are modified or deselected. Given the new set of observed symptoms, degraded mode can be exited and diagnostics can proceed with physical or functional assessment.

4. If a rectification is selected and performed during degraded mode assessment and no changes to the system are exhibited as a result of the functional check, the degraded module records that rectification, eliminates the performed rectification from consideration, and reranks the TOC's rectification list. The maintenance technician can then select another action from the TOC.

Hence, the diagnostic module can continue fault isolation and repair when faced with incorrect or missing data.

CLIPS IMIS-DM Redesign

The functional assessment module of the IMIS-DM, originally developed using C language, was thought to possess many characteristics and decisions better described within a logic-based expert system. Therefore, the functional assessment module was redesigned into an expert system using the CLIPS. This expert system tool written in and fully integrated with the C language was designed specifically to provide high portability, low cost, and easy integration with external systems. Hence, it was chosen as the language to describe the functional assessment module's logical decision process. The basic logic flow and the files containing the operating rules are depicted in Figure 8. Each rule within a file contains certain constraints for which the rule will activate or fire. The description of the logic flow is general; therefore, the formulas used to calculate parameters are not included. All of the calculations performed within the diagnostic module expert system are the same as the calculations described by Cooke et al. (1990).

The CLIPS version of the functional assessment module first obtains system symptom and fault information and provides the appropriate displays to select observed symptoms. This action is accomplished with rules within Get Observed Symptoms/getobsymp.clp. The next step in the diagnostic process is to obtain lists of faults spanned by the observed symptoms and to calculate failure rates from MTBFs for missing fault weights (Failure Rate Calculation/failrate.clp). After completing the failure rate calculations, spanned faults are grouped into fault combinations. Redundant combinations are eliminated. Probabilities are assigned to the nonredundant fault combination by the Fault Combination Probability Calculation/fltcmbprb.clp file. These combinations and associated probabilities are used to depict and rank rectifications and tests according to the Best Action and Best Test evaluations and are displayed to the maintenance technician (Rank Rectifications/newrctrk.clp, Rank Tests/newtstrk.clp, Display Ranked Options/disrnkop.clp). The Interleave Actions/newinterlve.clp routines are fired next, which in turn iteratively fires Secondstep Analysis/newscndstep.clp routines to produce an interleaved list of ranked tests and rectifications. After a list of five ranked options is completed, Action Choice/actnchc.clp rules are fired to display the interleaved list and provide the technician the opportunity to make an action choice from any of the current ranked lists. When an option is selected, the expert system uses forward chaining to search the entire rule base for a rule that will satisfy given constraints based on the outcomes of the action performed. In the event there is a change in symptom as a result of a rectification, rules will be fired to reinitialize the symptom/fault status and diagnostics will continue to isolate and repair by proceeding with the Failure Rate Calculation/failrate.clp. In the event no change in symptom occurs, fault combinations are reduced and their probabilities recomputed, continuing diagnostics at Fault Combination Probability Calculation/fltcmbprb.clp. This process continues until all symptoms from the observed symptom list have been alleviated.

The CLIPS functional assessment module, due to the overhead of C code used to produce the CLIPS logic-based environment, tended to be less memory efficient than the original functional assessment module produced in purely C code. Due to this memory overhead, the CLIPS functional assessment module was also slower than the original C-coded functional assessment module. Consequently, although the CLIPS module provided an excellent method for viewing the diagnostic logic, it has not been adopted in any formal application of the IMIS-DM.

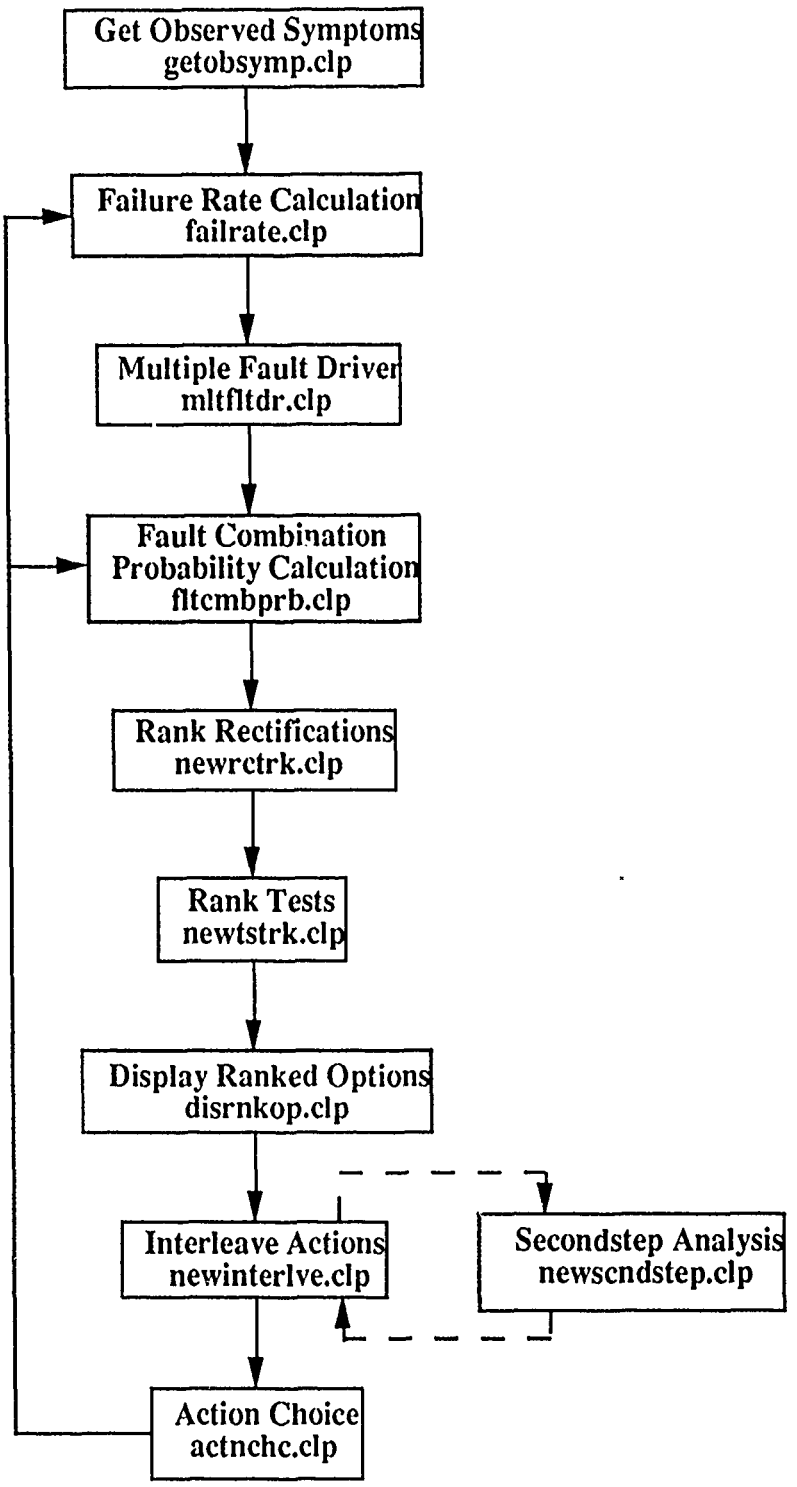


Figure 8. Functional Assessment Module's Logical Decision Process.

III. DIAGNOSTIC ENHANCEMENTS

The diagnostic enhancements described in this section provide additional capabilities to the diagnostic module itself and to the diagnostic presentation system. As a result, this section has been organized as two distinct categories: Enhanced Diagnostic Module Functions and Enhanced Diagnostic Presentation Capabilities. The enhanced diagnostic module functions provide the diagnostic module with a means to efficiently utilize information to isolate and repair more effectively, and to change results of incorrect data entry by the maintenance technician. The enhanced diagnostic presentation capabilities provide diagnostic presentations with a data validity check for test data entry, and give the maintenance technician the ability to feed back information to designate an activity as complete.

Enhanced Diagnostic Module Functions

The enhancements described in this section have created a more efficient and accurate diagnostic module. The module now performs degraded mode and revised critical fault assessment, and captures information gained from previously failed tests. It also considers "But Not" data entry and time saved for accessing groups of components or LRUs for testing and repair. Other enhancements to the diagnostic module allow the technician to change test and functional check outcomes in case of incorrect entry of results.

Failed Faults from Previous Test

Earlier versions of the diagnostic module sometimes lost failed test information. Initially, upon entering diagnostics, observed symptoms provide a set of spanned plausible faults of which there may be one or more true faults (multiple faults). Loss of failed test information occurred when there was more than one observed symptom with multiple faults and two tests were performed in sequence, leaving the plausible set void of any possible fault combinations. For instance, a failed test implicates a set of possible (plausible) faults in a multiple-fault situation; the remainder of the faults, including one of the true faults, is placed into the maybe set. The implicated faults remaining in the plausible set are tested again, isolating to one of the true faults and moving the remainder of the plausible faults to the maybe set. The fault remaining in the plausible set is known bad but no possible fault combinations can be created from that single fault in this scenario. Therefore, the faults from the maybe set are placed back into the plausible group to isolate and repair again. Previous test results may implicate a faulty component but because of recursive fault processing, the diagnostic module may not recognize that the fault is confirmed bad.

A new type of fault list is used in the enhanced version to correct this problem. This new list type is called the isolated faults list. Whenever the plausible fault list contains only one fault, the fault is placed in the isolated faults list before other processing is done. When the plausible set is rebuilt, the isolated faults list is searched for the first isolated plausible fault. If one is found, all other faults are moved to the maybe set. Because the fault is the only one in the plausible set, the module will recommend its rectification with a 100 percent probability.

Access Group

An access group is a group of components unveiled by removing a panel or cover. When ranking tests or rectifications, a diagnostic advisor should consider access group factors for rectification and testing time efficiency. Diagnostic efficiency may be gained when actions are performed on an access group which then reveals other fault-associated components; these components normally involve high access times but now these times are small. Previous IMIS diagnostic module versions did not consider access times in the ranking of tests and rectifications.

Access times in previous versions were assigned to each individual action and were not considered for a commonly accessible group of actions.

The method of approach used to develop this capability was essentially the same as that used to develop a Multiple Outcome Test (MOT) evaluation capability described in the report by Cooke et al. (1990). The access group algorithm is designed such that once access is gained, each test in the group can be accomplished as though no access time is required. In addition, the best test evaluation in these circumstances is merely an extension of the current best test algorithm. This feature was created by adding an enhancement factor to the best test that accounts for the additional fault isolation capability obtained by gaining access.

The enhancement factor used is:

$$EF_j = \frac{\sum_{k=1}^{N}\left[\frac{\sum_{i=1}^{PS} FR(1)}{FR(PS)} \times \frac{\sum_{i=1}^{PS} FR(0)}{FR(PS)}\right]}{N \times \Sigma T} \qquad (9)$$

Where EF_j = the enhancement factor for test j,

$FR(PS)$ = the sum of all the failure rates for faults in the plausible set,

$\sum_{i=1}^{PS} FR(1)$ = the sum of the failure rates for spanned faults for a given test (T) in the plausible set of faults,

$\sum_{i=1}^{PS} FR(0)$ = the sum of the failure rates for unspanned faults for a given test (T) in the plausible set of faults,

$\sum_{k=1}^{N}$ = the sum of the products of spanned and unspanned tests within the access group,

ΣT = the sum of all test times including access time for the group (e.g., for an access time of 10 minutes, creating access to three 5-minute tests, $\Sigma T = 25$), and

N = the number of tests in the access group.

The enhancement factor is set to zero if no additional tests are included in the access group. Hence, the best test algorithm used in the IMIS diagnostic module after this enhancement is:

$$BT = \max \frac{R_j \bar{I}_j}{T_j} + EF_j \qquad (10)$$

Where

BT = Best Test value,
R_j = sparseness ratio of the test span,
$\bar{I}_j$ = the average information gain, and
T_j = time to accomplish test j.

"But Not" Data Entry

The "But Not" algorithm implemented in the redesigned diagnostic module retrieves information from test results in the form of observed outcomes and spanned faults, and determines what faults are implicated and exculpated based on the test performed and outcome observed. Previous versions of the diagnostic module did not include "But Not" data entry logic when manipulating faults from test results. The exclusion of this "But Not" data entry logic resulted in inefficient fault isolation and repair decisions because known good faults remained under investigation.

The "But Not" algorithm development approach first identified what test outcomes would be returned from test results and how faults would be manipulated. Tests can have two or more outcomes. Every test has one pass outcome and at least one fail outcome. Pass outcomes only exculpate faults and if a pass outcome is observed, no fail outcomes can be exhibited. Fail outcomes, however, can implicate and exculpate faults simultaneously.

The "BUT NOT" algorithm processes two types of tests: (a) BINary (BIN) tests and (b) MOTs. The BIN tests are very simple tests that exhibit either a pass or fail outcome, whereas the MOTs are more complex. MOTs have one pass outcome and two or more fail outcomes. There are three types of MOT tests: (a) Complete And Enter One (CAEO), (b) Complete And Enter All (CAEA), and (c) Exit At First Failure (EAFF). The CAEO MOT is completed in full, and only one outcome can be entered upon completion of the test. However, CAEA MOTs are also completed in full but all observed outcomes are entered. EAFF MOTs are completed only to the point at which the first failure is observed, and at that point the observed outcome is entered.

Each outcome exhibited from a test result maps to a set of spanned faults, exculpated and/or implicated. BIN, CAEO, and CAEA tests have one pass outcome that, when observed, exculpates all faults spanned by the pass outcome. When a fail outcome(s) is observed from these tests, the diagnostic module implicates and exculpates all faults for the observed fail outcome(s) and then exculpates all the implicated faults of the nonobserved outcomes. The EAFF also exculpates all faults of the observed pass outcome. But, if a fail outcome is observed, the "But Not" algorithm exculpates all implicated faults for prior nonobserved outcomes in the performed sequence and implicates and exculpates faults of the observed outcome. Because of the implementation of the "But Not" data entry logic, known good faults are exculpated while suspected faults are evaluated and the "bad" fault(s) are isolated and repaired.

Account for TOC Actions

Earlier versions of the diagnostic module did not effectively allow choices to be made from the TOC. This limited the maintenance technician's ability to perform tasks which he considered pertinent but were not in the interleaved actions list. Accounting for TOC actions was implemented easily within the diagnostic module. Now, whenever a TOC test or rectification is chosen, the diagnostic module needs only to be informed that the action selected was not from the interleaved actions list. If a test is selected, the observed test outcomes need to be passed to the diagnostic module as well. Furthermore, if the task pertains to any of the existing diagnostic groups, that group will be updated. If no appropriate diagnostic group exists, one is created.

Change Test Result

If a maintenance technician erred in selecting or entering a test outcome, earlier diagnostic module versions would not allow easy correction. In the redesigned diagnostic module, a previous actions list is used to provide a means of correcting an entry. The previous actions list contains a list of all the functional checks, tests, and rectifications previously performed. It also keeps a copy of the machine's diagnostic state before the action. Once a previous test is selected, the new results are passed to the diagnostic module. The diagnostic group pertaining to that test is removed and replaced with the copy stored with the previous action. If no diagnostic group exists, the copy stored with the test is added to the list of diagnostic groups. The new test results are passed to the diagnostic group. The next item in the previous actions list belonging to the same diagnostic group as the test is updated with the new diagnostic group. Then the diagnostic group is updated with the results of the action. This process is repeated until the previous actions list is exhausted.

Revised Criticality

The criticality function has been designed for situations when operational demands prevent maintenance practices that take a minimum time to repair a malfunctioning system. Under some circumstances, it is possible operational requirements would declare a system usable if it can be determined the fault present in the system is in a part of the system not essential for the next sortie. This circumstance is recognized in the Air Force by the current practice of declaring a system Partially Mission Capable (PMC). When this practice is invoked, the criticality function of the IMIS-DM can be set to accommodate a change in maintenance practices.

The diagnostic data allow faults to be assigned to certain systems and subsystems that may or may not be designated as critical. If a system or subsystem has been declared critical, then the diagnostic module is modified to make recommendations based upon a criticality algorithm. The criticality algorithm is designed to allow a critical fault decision at the earliest possible time in the maintenance process. At each step of the process, the algorithm attempts to answer the following question: "Does the fault in the system lie in the group of faults declared critical?" If not, then the weapon system can be declared ready to operate. If so, then the critical fault can be isolated in the most efficient fashion using normal diagnostic algorithms.

In developing the algorithm, it soon became obvious the hardest part of the problem involved evaluating available tests to determine if the above question could be answered as a result of a single test. Furthermore, if the answer to the question was affirmative, then how far did the session have to deviate from a truly efficient diagnostic path in order to obtain the answer? In developing the algorithm, the following definitions were posited.

1. Critical Test - A test that examines all potential faults declared critical.

2. Critical Rectification - A rectification that acts to repair all faults declared critical.

Obviously, if a critical test passes, then the system can be declared ready for operation without any further repair actions. Further, if a critical test fails, then diagnostic efforts can focus on the critical faults in a system without further attention to possible system faults that lie outside the set of critical faults. However, it is very possible a test that examines all of the critical faults will be quite inefficient should a fail result occur. That is, a critical test that fails may result in an implicated set of faults far larger than the set of faults that would have been implicated by a better test selection based upon the entire system's potential fault state. The problem, then, was to find a methodology maximizing the benefit to be gained from passing a critical test yet minimizing the costs associated with failing a critical test.

When the critical set of faults comprises less than half of the total set of plausible sets, failing a critical test will result in isolation of the critical fault in fewer steps than would have been required to isolate the same fault had the test chosen been one which split the total plausible set in half. If, however, such a critical test had passed, then the fault would have remained in the system during the critical sortie, and the fault would have had to be repaired at some future time when criticality is not an operational consideration. In such a case, the total time to isolate and repair the noncritical fault will be greater than that required had a half-split test been accomplished. The methodology chosen to implement the criticality decisions in the IMIS-DM considers these aspects of the problem. The following assumptions apply in the development of the criticality algorithm:

1. Operational considerations can result in the imposition of critical requirements.

2. The faults comprising a critical set for a given critical function can be identified in the data base.

3. The value of an operational hour to the operations community is at least as great as the value of a maintenance labor hour to the maintenance community.

4. The tests available in the critical set of faults are at least as effective as the tests available in the full system set of faults.

Information theory shows the minimum number of steps to fault isolation in a given set of faults is expressed by the base 2 logarithm of the number of faults in the plausible set (Log_2 #F). Consequently, the minimum number of steps to isolate and repair a fault in a system can be expressed as:

$$STREP = 1 + Log_2 \#F. \tag{11}$$

If a critical test passes, then the system can be declared operationally ready for the next sortie. If it fails, then the fault is known to be contained in the critical set of faults, and the steps to isolate and repair in the critical set can be declared similar to Equation 1 and expressed as:

$$STFLY = NCF/\#F + \{[1 + Log_2 (2*CF)]*CF/\#F\}. \tag{12}$$

Finally, the fault must eventually be repaired when criticality is nc longer an operational consideration; so, the total steps to repair the fault, including the actions both during and after criticality considerations, can be expressed as:

$$STTOT = \{[1 + Log_2 (2*CF)]*CF/\#F\} + \{[1 + Log_2 (2*NCF)]*NCF/\#F\}. \tag{13}$$

The advantage to operational considerations is:

$$ADVOPS = STFLY - STREP. \tag{14}$$

The cost to total maintenance steps is:

$$STMAIN = STTOT - STREP. \tag{15}$$

The decision equation for whether the critical test should be preferred over a conventionally chosen test can then be expressed as:

$$\text{When } Val*ADVOPS \geq STMAIN, \text{ do the critical test.} \tag{16}$$

Where

Val	=	the value of an operational hour over a maintenance hour,
STMAIN	=	the cost of total maintenance steps,
ADVOPS	=	the advantage to operations of performing a critical test first,
STTOT	=	the steps to perform maintenance required to bring the system to operational condition after a critical test,
STFLY	=	the steps to declare a system ready for operations when a critical test is performed as the first test,
STREP	=	the steps required to repair a system when the most appropriate test is performed first,
#F	=	the number of plausible faults in the system,
NCF	=	the number of noncritical faults in the plausible set of faults, and
CF	=	the number of critical faults in the plausible set of faults.

The above equations were subsequently modified to account for the probability of the fault sets and multiple outcome tests. Each test's outcomes can create a fault set. For the pass outcome of each test, the fault set will be the unspanned plausible faults of the test. For a fail outcome, a fault set will consist of the plausible spanned faults of the outcome. To obtain the probability for the pass outcome fault set, the probability of the plausible spanned faults is used. For fail outcome fault sets, the probability of the outcome will be used.

The steps to isolate faults in each fault set are conditioned using the probabilities formulas. The end result is the sum of the steps for the pass outcome and all of the fail outcomes. The final forms of the equations in the criticality algorithm of the IMIS-DM were:

$$STREP = \left[(1\text{-}Pfb) * Log_2(2*\#nsb)\right] + \left[\sum_{i=1}^{\#fob} Pfob_i * Log_2(2*\#sfob_i)\right] + 1 \qquad (17)$$

$$STFLY = \left\{\sum_{i=1}^{\#foc} \left[Pfoc * Log_2(2*\#sfoc_i)\right]\right\} + 1 \qquad (18)$$

$$STTOT = \left[(1\text{-}Pfc) * Log_2(2*\#nsc)\right] + \left[\sum_{i=1}^{\#fob} Pfoc_i * Log_2(2*\#sfoc_i)\right] + 1 \qquad (19)$$

Where

Pfb	= the probability the best test will fail,
#nsb	= the number of plausible faults nonspanned by the best test,
$Pfob_i$	= the probability of fail outcome i of the best test,
#fob	= the number of fail outcomes for the best test,
fob_i	= fail outcome i of the best test,
$\#sfob_i$	= the number of plausible spanned faults from outcome i of the best test,
Pfc	= the probability the critical test will fail,
#nsc	= the number of plausible faults nonspanned by the critical test,
$Pfob_i$	= the probability of fail outcome i of the critical test,
#foc	= the number of fail outcomes for the critical test,

foc_i = fail outcome i of the critical test, and
$\#sfoc_i$ = the number of plausible spanned faults from outcome i of the critical test.

After calculating the results from Equations 17 through 19, Equations 14 through 16 are employed to arrive at a decision as to whether the critical test will be recommended.

Enhanced Diagnostic Presentation Capabilities

The enhancements described in this section have created a more efficient and accurate diagnostic presentation environment for the maintenance technician, with a data validity check on test outcome information to control entry of incorrect or out-of-bounds test values and a feedback entry for the maintenance technician to indicate an unsuccessful or incomplete maintenance action.

Change Functional Check Result

In previous versions of the presentation system, the technician could not change the results of a functional check. There were two reasons for this problem. First, the diagnostic module did not possess the capability to change results of any test. The capability to modify a test's results has been added and is described in detail under the Change Test Results paragraph of the Enhanced Diagnostic Module section. The second reason for this problem was in the presentation module. When a functional check passed, the presentation module assumed diagnostics were complete and exited the diagnostics routine. This problem has been corrected by only allowing the technician to exit diagnostics. The capability to change the results of any test is accessible from within diagnostics. As a result of the above actions, the ability to change the results of a functional check is now available.

Data Validity Check

Upon performance of a test, feedback on test outcome results is entered into the IMIS-DM either manually or automatically. In previous versions, no test data validity checks were available to the technician to designate whether a particular test result was within the expected values of an acceptable test result. For instance, a particular voltage check (test) should result in 5 +/- .01 volts for a pass. A legitimate fail would be from 0 to 4.989 volts or 5.011 to 10 volts, indicating the acceptable range of test voltage values is between 0 and 10 volts. In previous versions, if the maintenance technician tested the incorrect wire in the bundle and returned a value of 24 volts from the check, the options would be to pass or fail the test. After making a test entry, the presentation module would accept the word of the technician that the test was performed properly and the results were correct. This test entry was then presented to the diagnostic module with no test data error checks, and in doing so, diagnostics proceeded down the wrong path to fault isolation and repair.

Hence, as a result of this investigation, the presentation module was equipped with a data validity check. The data validity check retrieves particular test outcome ranges from the CDM data base, checks the test outcome ranges against actual values from the manually or automatically entered test result, and accepts only values within the expected realm of the test outcome. If the actual values returned from the test result are not within the variance of expected values, an appropriate message is displayed to the technician.

Maintenance Actions

Maintenance actions are rectifications that do not remove and replace (R&R) components but rather, adjust components. Previous versions of the presentation module treated each maintenance action as an R&R and did not consider instances when a repair or an R&R would be required if the maintenance action were unsuccessful or could not be performed properly. For instance, consider what would happen if a maintenance action on a component could not be performed successfully and the presentation module acknowledged only R&Rs. First, the presentation module would require a functional test even though nothing was fixed. Then, the diagnostic module, being given incorrect information on the outcome of the maintenance action, could suggest another maintenance action on the same component, perform an R&R on another component (ignoring the faulty one), or perform further unnecessary tests on the faulty component or other components.

A maintenance action used as a rectification presents the unique situation of a passed test requiring a system health check. The reason is that the maintenance action, if successful, was a rectification (with a system health mapped as its conformation test). But, if the maintenance action was unsuccessful, it mimics a test that implicates a set of faults. As an example, Table 4 shows a system with potential faults of Out Of Alignment and Will Not Align among its set of manifested failures. The Out Of Alignment requires an alignment maintenance action whereas the Will Not Align requires an R&R. If the align rectification is accomplished, its success must be determined before proceeding. The test mapped to the Out Of Alignment is, "Was the maintenance action successfully completed?" If the answer is "yes," then the system health check must be performed to ensure the Out Of Alignment fault was the fault present in the system and the alignment did in fact remove the Out Of Alignment fault. If the answer is "no," then the fault Will Not Align is implicated and diagnostics and repairs associated with that set must be performed. A third option occurs if the alignment was started but could not be completed. In this case, both the Out Of Alignment and Will Not Align faults are implicated and the repair actions required by these faults are indicated. Hence, each of these outcomes obviously produces different sets of implicated and exculpated faults and system health information.

Therefore, the logic to handle this unique situation requires data element modifications and presentation software modifications. Within the CDM rectification data elements for maintenance actions, the author is required to list both tests (aligned and system health) in sequence to prove that the Out Of Alignment fault was at fault and that the maintenance action did fix the problem. The first test is an MOT similar to the example below or a binary test. The second is the system health and is recommended if the first test passes.

IV. POTENTIAL FOR ANALYSIS OF DIAGNOSTIC DATA

The availability of the IMIS-DM, the data it requires, and the data it can provide through automatic data collection extend significant opportunities and implications for engineering and support analysis. A preliminary examination of these opportunities was conducted during this task. Several important issues identified are discussed in this section. Although the interface between IMIS and the Core Automated Maintenance System (CAMS) is not yet fully defined, we have assumed for this discussion a CAMS or a CAMS-like facility will be present in the future maintenance environment. The maintenance hierarchy, from the flight line to the depot, should have access to the maintained system's CDM format diagnostics models and data and the previous maintenance actions taken, including initiating symptoms, relevant in-flight data, time-stamped keystrokes from the Portable Maintenance Aid (PMA), action taken, and so forth.

Table 4. Maintenance Action Test Example

Select the output that represents the results of the maintenance action.			
	Completed?	Successful?	
Outcome #1	yes	yes	(pass)
Outcome #2	yes	no	(Fail 1)
Outcome #3	no	---	(Fail 2)
Outcome #1	Exculpated both faults -- Out of Alignment and Will Not Align		
Outcome #2	Implicates fault -- Will Not Align		
Outcome #3	Implicates both faults -- Out of Alignment and Will Not Align		

The maintenance action is mapped to Out of Alignment while the repair is mapped to both Out of Alignment and Will Not Align.

Timing and Functional Allocation

At this writing, the functions of a fully implemented IMIS are not fully defined, especially the feedback analysis functions. Allocation issues, as opposed to functional issues, needing definition are those such as whether the postmortem analysis of a failed fault isolation sequence should be instituted immediately at the flightline or documented for later scrutiny at the depot (i.e., should a failure of the fault isolation process be subject to immediate or deferred maintenance?). The four candidate functional times and/or sites for IMIS field feedback analysis are (a) during on-equipment maintenance activity; (b) off of the equipment at the intermediate shop; (c) data analysis at the wing level; and (d) at the depot. The advantages and disadvantages of each are discussed below.

During On-Equipment Maintenance Activity

Flightline data analysis will be of no advantage if it is confusing, time-consuming, or requires an inordinate level of user sophistication and training. A great deal of data collection and analysis can be hidden from the maintenance technician. The on-equipment IMIS TO, communication, and diagnostic functions will be supported and enhanced by aggregate performance statistics such as Reliability and Maintainability (R&M) and aircraft Not Mission Capable (NMC) rates, although these analyses need not be performed at the flightline. Real-time data review and analysis that may be beneficially introduced at the flightline are the review of recent maintenance history and simple trending to interpret these data. Trend analysis may also be beneficially performed on an IMIS on-equipment or Aircraft Maintenance Unit (AMU) workstation with access to the CAMS data. The extent to which it is possible and desirable to introduce trend analysis to the flightline technician is an empirical issue.

The automated forms generation system to debrief the maintenance technician immediately after performing a maintenance job should eliminate ambiguity in subsequent interpretation of successful IMIS diagnostic sessions. The user interface should allow the technician to note where the TO data do not match the job or equipment, although this facility is of limited use in troubleshooting.

Off-Equipment at Intermediate Shop (I-Shop)

Running real-time data base searches and analyses would benefit the intermediate shop level by identifying and eliminating intermittent failures, as well as increasing the confidence in all other maintenance. Additionally, some of the systems IMIS supports might require off-equipment flight data screening. The maintenance organization IMIS on-equipment workstation, the Aircraft Maintenance Unit (AMU) workstation of the future, however, will have neither the sheltered environment of a present-day intermediate shop to work from nor the ability to perform much off-equipment work on electronics equipment beyond screening components before condemning them. Hence, most of the maintenance and diagnostic data base manipulation and analysis functions will be developed for depot-level use. Should the intermediate shop still be an operational requirement on the next generation of aircraft, the depot automated functions could be readily adapted for the intermediate shop role.

At Wing Data Analysis

The benefit of providing maintenance information beyond R&M, availability, and sortie generation rate to the wing-level decision makers is not clear. Statistics such as "percent of on-hand equipment containing intermittent faults" or "percent of unnecessary removals attributable to maintenance inefficiencies" could be derived at the wing level, but interpretation and productive use of the numbers are difficult. The additional insight available from a depot fleetwide perspective, as well as the undesirability of increasing wing manpower overhead, suggests that the implementation of advanced maintenance analysis functions be at the depot. On the other hand, on-site, wing-level personnel are better able to interpret the derived statistics. If analysis is centralized at the depot, a liaison activity will likely be required at the wing-level data analysis shop. The point at which it is advantageous to implement wing-level-specific analysis at the wing, as opposed to at the depot, depends on the degree to which this on-site attention is required.

At Depot

Feedback analysis at this level can support depot engineering, TO maintenance, and diagnostic module maintenance. The major drawback is the distance and time delay inherent in a centralized depot operation. The major error of the past in dealing with this separation has been communication; certain information, such as test results from previous levels, is not transmitted, resulting in the Air Force's inability to assess this information's value to guide the information system redesign. If adequate maintenance data are not captured by the field maintenance data systems, we will continue to experience often hidden inefficiencies in cycling equipment through the maintenance system.

Depot analysis will consist of screening and aggregating the maintenance traces to determine action taken and hence R&M statistics. The diagnostic data module issues of when to update and whether to maintain separate data bases for separate locations are discussed below. It is anticipated that the bulk of the data manager's time will be spent on configuration control issues and in engineering support tasks such as testability assessment and sensitivity analysis.

Data Base Interfaces

The interfaces between the CDM and the IMIS-DM and the other information systems within the depot-level support environment need to be defined in terms of the functional requirements. The systems with which IMIS must interface are also undergoing definition at this time, making these interface recommendations tentative.

CAMS

CAMS data should include How Malfunction (HOW MAL) and Action Taken codes for all maintenance actions. It can be assumed that complete IMIS use information can be passed to the depot from previous levels of maintenance. Additionally, relevant in-flight data and Built-In Test (BIT) results automatically captured during the Portable Maintenance Aid (PMA) use should be passed through the information process. Similar maintenance data from intermediate- and depot-level maintenance need to be collected. Documentation of any irregular diagnostic function should include actions attempted when the normal diagnostics failed, all available in-flight data, and observations from participants on possible diagnostic module or process problems.

Engineering Activities

An engineering data base containing several alternative representations (geometrical, functional, net list) of the target weapon system is the most likely engineering environment the IMIS and the IMIS feedback analysis will need to support, at least for emerging systems. The major interface between the IMIS diagnostic data and the engineering data is in failure modes and effects, and testability data and analysis. The major interface between the IMIS procedural data and the engineering support community is through the system's Reliability-Centered Maintenance (RCM) program and associated task analysis activities and data, which, in turn, connect the IMIS diagnostic data to the TO authoring and training communities.

The first step in manipulating the IMIS diagnostic module for the benefit of depot support engineering will be to update the diagnostic description of the equipment. The original intent of the diagnostic description portion of the IMIS CDM module was to serve as a common functional description of how equipment fails, how it behaves under failure, and the resources available to deal with failures. This diagnostic description convention was intended to integrate data from failure modes and effects analysis, BIT and Automatic Test Equipment (ATE) test set descriptions, and other functional descriptions such as dependency models from testability analysis. Thus, the maintenance traces and reports first need to be analyzed, and the data base updated to reflect maintenance experience.

The analysis of this data base to support engineering should be identical to a testability analysis. Changes to reliability and task time data will contribute to maintenance performance statistics such as expected man-hours or numbers of failures. Additional failure modes will necessitate a reevaluation of fault coverage. This check will afford the capability to evaluate the system's estimated performance characteristics' sensitivity to additional tests, changes in reliability or task times, and procurement of additional support equipment or spares. The effect of this new capability will be a change from support engineering's current emphasis on reliability improvement programs to a consideration of the investment alternatives including reliability improvement, equipment or test equipment redesign, and procurement of additional spares or support equipment.

The interface between the IMIS procedural description TO data and the engineering support community is through impacting the system's RCM program and associated task analysis activities and data. The future task analysis environment will consist of a loose framework of automated Computer-Aided Design tools supporting task identification, task analysis and validation, virtual prototyping, safety analysis, and the development of automated procedural TOs. The IMIS will necessitate the review of procedures when predicted performance parameters disagree with achieved field statistics, as well as the generation of procedural TOs when new procedures are required to support field maintenance.

Data Analysis

Assign Action Time (AT) and HOW MAL Codes

Current organizational maintenance practice and the IMIS design are centered upon AT and How Mal codes reporting. If anything beyond passive reporting is required of the maintenance technician, items in these categories should be collected during the automated forms generation or maintenance debriefing. Field experience would help the design of an appropriate user interface.

It is possible to validate reported AT and HOW MAL codes as an internal validity check through analysis of the fault reporting system data, supply system data, the on-board test interface function, and the PMA keystroke trace. This processing should be done at the end of a maintenance job, so any required information can be provided by the performing maintenance technician. Activities requiring clarification are (a) deviations from expected task times, (b) nonrelated part or spares consumption, and (c) requests for nonrelevant technical data. This restriction results in the transmittal and review of relevant data only at subsequent levels, which increases system efficiency.

Update Reliability

Fault Weights. Once the AT and HOW MAL codes are determined, component reliability is readily computed. To maintain consistency with current reliability reporting practices, the IMIS-DM must consider component reliability the basic modeling construct but must also compute fault weights for each identified fault within a component for use in field diagnostics. In those cases where a component is replaced without complete fault identification, it is necessary either to examine the next level of maintenance's fault isolation trace or to divide the failure rate among the several still-plausible failure modes. The former practice can be estimated by simulation but is impractical.

Mean Time Between Failure (MTBF). MTBF can be determined from the maintenance diagnostics trace data. In those situations where removals are performed without complete isolation to one component, there is an increased requirement for separate reliability measures. The distinction between MTBF for true failures and mean time between removals may become valuable to consider in discussing reliability. For instance, LRU manufacturers are required to build to a specified MTBF for true failures. However, if inadequate diagnostic capabilities lead to unnecessary removals, then the impact on the maintenance community is much the same as if the MTBF had not been achieved.

Quantity Per Application (QPA) Issue. Finally, multiple installations of a single item within a weapon system must be considered. The fault isolation and support engineering benefits of monitoring failure rates of separate installation locations will produce benefits in increased field diagnostic efficiency and engineering support decisions. However, the cost for these advantages is the need for separate part tracking for the affected components.

Update Other Parameters

Several other parameters will need to be updated based upon field data analysis. Generally, these measures are less critical or accurate than reliability and task frequencies.

Symptom Frequency and Dominance. The observed frequency of symptoms, where many faults can produce multiple symptoms, may be difficult to predict prior to initial field experience. Because of operational policy or conditions and the impact of some symptoms, additional diagnostic efficiency may be gained by exploiting the empirically gained probability of a certain fault occurring, given a symptom, rather than assuming that all of a fault's symptoms are equally

likely. The frequency of symptoms will need to be evaluated and updated; this task will be relatively simple because this information is generally captured as part of pilot debriefing or inspection. The frequency information could also be of use in reducing the size of a data base and in initial or proficiency maintenance training planning; it will also impact logistics requirements, if testability is used to estimate test and support equipment and spares.

Activity Time. Activity times will be highly variable because of uncontrollable contingencies and interruptions in the workplace. Some connection between activity times and engineering standards to link the activity times to traditional engineering task estimation techniques needs to be developed, such as the model activity time. However, this issue may be of little importance since the IMIS diagnostic activity is composed of a number of tests, access, rectification, and verification tasks that need be only ordinal level measures.

Access Time. There is little practical use for access time estimates outside of the test sequence determination of the IMIS diagnostic module. Possible support engineering uses would be in determining the time benefits of rearranging equipment or in decreasing access times. Situations where a second Air Force Specialty (AFS) is required to perform facilitating research might be examined with the idea of expanding the prime AFS's duties to include the facilitating AFS's task.

The only potential problem occurs if bias exists systematically between maintenance actions. Though it is unlikely that R&R action A will be systematically under- or overreported relative to R&R action B, policy can produce systematic differences between classes of activities. For example, one location might charge Not Mission Capable/Supply (NMCS) times to an R&R time, or the times for an enabling maintenance action might not include the waiting time for an appropriate AFS to arrive for task performance. The blanket analysis of maintenance traces without consideration of these issues will result in updating the data base to recommend inefficient diagnostics.

Support Equipment Availability. Activity times for obtaining support equipment should be examined over time and across locations and systems to determine support equipment usage and the impact, if any, of differing levels of support equipment. The results can then be used to update support equipment allocations and entered into the support engineering investment decision matrix to compete with the reliability improvement programs for funds.

Remaining Life Prediction. Mechanical systems wear out. Monitoring the wear and predicting when the equipment will become unserviceable has been the traditional diagnostic focus of the engine and hydraulic communities. The additional modeling construct required is a remaining life failure mode, which gets updated by the inspections defined for the particular component. This mode is in addition to the same type of fault seen in equipment that exhibits Poisson failure mechanisms requiring rectification prior to return to service.

False Removals

The unnecessary removal rate has traditionally been a measure of maintenance inefficiency. Two points need to be made on this topic. First, the presence of ambiguity groups in equipment entails a certain amount of component replacement be done to fault isolate; some intrinsic, unnecessary removal rate is designed into the equipment. The distinction between false removals (where a good component was removed as a normal maintenance function) and unnecessary removals (where a good component was removed unnecessarily) is worthwhile.

Second, despite attempts to remove it from design, some decisions in equipment maintenance are not automatic and call for human judgment. Thus, the maintenance technician is forced to weigh possible unnecessary removals against leaving failed components on the aircraft. He should err toward unnecessary removals. On the other hand, the intermediate shop chief must choose between keeping possibly faulty components in his local inventory or shipping off good components. Though not as clear a choice as the flightline technician's, the intermediate shop chief should be reluctant to part with possibly good components. Thus, most of the diagnostic errors in the system should accumulate between the flightline and the intermediate shop. This conclusion results in visible parts cycling but is considerably preferable to the less visible alternatives of leaving failed components on aircraft or inserting a significantly larger proportion of available spare parts into the depot repair pipeline.

Determining Diagnostic Model Errors

Inconsistencies between field experience and expected maintenance behavior point to errors in the technical data, the diagnostic module, the data reporting and collection system, or the software for any of these. Evidence for error in the diagnostic module in particular is found in systematic inconsistencies between observed field behaviors and predicted behaviors. These inconsistencies can be found in the analysis of failed diagnostic sessions; in differences between the observed and achieved performance parameters, numbers of actions, and spares consumption, etc.; and in maintenance outcomes not coherent with the maintenance trace or the module-derived process flows.

Causes of Parts Recycling

There are six plausible causes of parts recycling: (a) the intrinsic recycle rate, (b) "swaptronics," (c) misunderstandings, (d) inconsistency, (e) intermittent faults, and (f) errors in the diagnostic data.

The intrinsic recycle rate results from the presence of ambiguity groups within the equipment. The minimum false removal rate can be determined from testability analysis using the exhaustive look-ahead feature of the diagnostic support tool, choosing tests at each node while tests are still available. Another exhaustive look-ahead using the most favorable action at each node will produce an estimate of the likely wrongful replacement rate following the optimal strategy.

The practice of "swaptronics" is replacing components as a fault isolation strategy. This practice is discovered by comparing the maintenance traces (once screened for consistency) with the minimum and most likely wrongful removal rates to determine whether the obtained recycle rate differs significantly from these. If the maintenance force is familiar with the equipment and the maintenance activity, the obtained recycle rates should reflect a test cost that is consistent with the time and effort of performing that replacement in terms of the range of productive maintenance actions available at each node. The acceptability of these decisions against the actual relative costs of the available maintenance actions must be judged case by case. Improvements in the troubleshooting costs may be achieved by training or changing the maintenance environment to make the less costly options more attractive (as through redesign, additional support equipment, etc.).

Misunderstanding of, or errors in, the TOs will cause parts recycling. These misunderstandings will show up as inconsistencies between the model's expected behaviors and the maintenance traces. Generally, inconsistency between the expected fault isolation flow and observed maintenance traces indicates lack of training; consistent differences suggest erroneous procedures, model data, or consistent errors in training. These problems can be solved by training, rewriting the TOs, or correcting uncovered errors in the equipment model.

Inconsistency between what is tested at different levels of maintenance should be apparent through testability analysis. If inconsistent results are still noted, engineering needs to resolve the problem.

Intermittent faults are faults that have no ground test associated with them or faults whose symptoms are not stable or reliably observed. Testability analysis will avoid these problems; the problem of mapping airborne symptoms-to-faults must be resolved at the design level. The major avenue for handling these problems is in-flight data collection. Without these, statistical analysis can be used to suggest areas for scrutiny but, due to the unreliable nature of the problem, changes to maintenance procedures alone based upon actuarial strategies will inherently have large mistaken removal rates.

Errors in the diagnostic data also contribute to parts recycling. Failure of an unambiguously isolated rectification to fix a failure indicates the presence of either an unknown fault, a missing symptom-to-fault mapping, an incorrect fault-to-test result mapping, or an incorrect fault-to-rectification mapping. Although it may, in principle, be possible to distinguish among these cases on the basis of the maintenance action trace and subsequent events within the maintenance arena, the preferred action to be taken in the event of a failed maintenance diagnostic sequence should include an immediate engineering examination of the circumstances surrounding the event and the development of a recommended fix.

Repair Does Not Fix Aircraft as Expected

The plausible causes of a failure of a fault isolation procedure to correctly identify the faulty item are misunderstanding, intermittent faults, and errors in the diagnostic data. Again, the recommended action is to focus immediate engineering attention on the problem within the diagnostic data.

Individual Differences

If differences among operating locations, time periods, aircraft tail numbers, and maintenance technicians exist, and positive value can be realized by analyzing these differences and maintaining separate maintenance data bases for these, the data analysis will need to distinguish among these entities where real differences exist. As in the case of estimating population statistics, there is no good substitute for a hands-on understanding of the system and its maintenance operation.

Among Operating Locations

The problems inherent in the data analysis problem over an entire fleet are exacerbated by the attempt to identify real differences among subgroups of the population, as correspondingly less data are available to establish parameters for the subgroups. For instance, many maintenance actions occur only two or three times each month at a single wing. Trying to determine wing differences on a monthly basis would be foolish.

Generally speaking, the mapping between faults and rectifications, test outcomes, and symptoms will be consistent throughout the fleet, with, of course, due respect for the problems associated with configuration control. However, one expects to see differences among reliabilities and, to perhaps a lesser extent, activity times.

Among Time Periods

The statistical problem here is to attempt to distinguish random fluctuations from genuine changes to the parameter of interest. In practice, some critical ratio between the parameter's difference or observed movement to the parameter's historical variability is chosen as representing a "real" shift for which the parameter is updated. The customary engineering models for updating parameters is to use a smoothing technique or a filter. The differences between the two approaches are that, as a general rule, filtering is implemented on a more sophisticated level. In actual practice, the schedule of parameter updates will probably be more profitably driven by the cycle of data updating in general. If data are updated monthly or quarterly, it is doubtful much will be gained by updating parameters more frequently.

Cyclical trends--as, for instance, those with a periodicity of 1 year due to seasonal changes --can be tested for after sufficient time has elapsed.

Among Aircraft

The questions of interest concerning differences among aircraft are of the sort that can be addressed by presenting a maintenance history of the particular aircraft and allowing the technician to synthesize the historic information with the standard diagnostic data.

It should be feasible to maintain separate data bases for separate operating locations. If the system specification calls for an examination of the separate locations, this should be no more trouble than using system-wide parameters for the fleet and keeping the base data analysis option open for system implementation.

Two effects should be considered: environmental and operational. In addition to location-by-location comparisons, we might want to examine explanative environmental and operational factors. A regression analysis of these explanatory factors against maintenance and operational criteria could be a useful tool in evaluating existing operations and in planning for future force support.

Among Personnel

Determining differences among personnel through analysis of maintenance traces for qualitative evaluation or adherence to TOs is a feasible activity yet but would not be popular, and probably not very valuable. Determining whether to implement analyses of individual differences is a policy question involving issues beyond the benefits to the fault isolation process.

However, it will also be possible to passively collect statistics on individuals regarding relative maintenance task performance times, spares consumption, and accuracy through the maintenance aid. These statistics could be of great value in providing objective criteria against which to determine training requirements.

Depot Support Engineering

The major tie between IMIS and support engineering is in the assessment of the impact of new failure modes, and changes to the test and symptom implications. Periodic reevaluation and respecification of the diagnostic testing system will maximize field experience and continuously improve equipment maintenance.

The current support engineering investment strategy is a periodic evaluation of the least reliable, most expensive components for reliability improvement programs. The adaptation of

testability analysis techniques will allow an assessment of the effects of changing support concepts, of buying more spares or support equipment, and so on, as well as calculating the benefits of reliability improvements.

V. CONCLUSIONS AND RECOMMENDATIONS

The IMIS-DM integrates many diagnostic theories which may be applied to effectively and efficiently fix aircraft systems. These theories provide the basis for incorporating information into the overall diagnostic decision-making process and utilizing the information to its fullest extent when analyzing, displaying, and repairing an existing problem.

R&D efforts in the diagnostic arena developed an IMIS-DM with the capability to provide basic diagnostics for faults caused by physical events resulting from temperature extremes, release of fluids, and physical abuses including aircraft battle damage. However, further research is needed to identify migration of physical events by identifying and defining the dynamics of hazards. Once the dynamics of given hazards are identified, we can develop models to simulate migration of each hazard within a three-dimensional system and identify the affected areas and components. Furthermore, the CDM must be modified to include both known hazard/fault relationships and mapping, as well as information necessary to perform migration models such as a three-dimensional coordinate system of aircraft systems, componentry, access groups, compartments, and so on.

One of the most important efforts to upgrade the diagnostic module involved an OO redesign of the C-programmed diagnostic module (now called the functional assessment module). This redesign effort created a diagnostic module compatible with the hierarchical structure employed by the CDM and created a system that provides rapid prototyping for future diagnostic enhancements and the interface necessary for the physical assessment module.

Enhancements as a result of previous research allow the diagnostic module to function effectively under less than adequate data availability conditions with the implementation of the degraded mode module. Other changes have provided tools to the technician to change the results of erred test and functional check results and to select actions from the TOC. Revised criticality, failed faults from previous tests, access groups, and "But Not" data entry have aided time to repair efficiency by including more maintenance environment information and ensuring better utilization of all information during the diagnostics decision process.

Enhancements to the presentation module include the data validity check to inspect test values returned by the maintenance technician in case incorrect procedures were followed during testing or incorrect measurement was taken, and the implementation of an effective means to approach maintenance actions.

Besides redesign and enhancement efforts, research into diagnostic data analysis, functions, and parameters has defined requirements to be collected by the diagnostic module for maintenance environment support. Analysis definition considered the parameters that could be collected by the diagnostic module and processed off-line to provide the reliability and repair information required to update the CDM data base for future diagnostic, depot maintenance, and logistics activities. Some of the feedback functions and parameters are:

1. Reliability - fault weights, MTBFs, and QPA
2. Maintenance - AT and HOW MAL codes, symptom frequency, action times, and support equipment availability

3. Repair feedback - multiple faults, new faults, CNDs, false removals, and scheduled maintenance

Although the analysis functions have been defined, full development and implementation of these functions has not been performed. Continued R&D efforts are required to give the diagnostic module the ability to capture, analyze, and transfer the appropriate information to off-line sources (e.g., CAMS and the IMIS workstation).

REFERENCES

Air Force Human Resources Laboratory. (1989). Draft specification for digital technical information. Wright-Patterson AFB, OH: Logistics and Human Factors Division, Air Force Human Resources Laboratory.

Cooke, G.R., Myers, T.A., Jernigan, J.H., Maiorana, N.A., Mason, D., & Link, R. (1990). Integrated Maintenance Information System (IMIS) diagnostic module, version 4.0. (AFHRL-TP-90-13, AD-A222 333). Wright-Patterson AFB, OH: Logistics and Human Factors Division, Air Force Human Resources Laboratory.

LIST OF ABBREVIATIONS

AFHRL	-	Air Force Human Resources Laboratory
AFS	-	Air Force Specialty
AMU	-	Aircraft Maintenance Unit
AT	-	Action Taken
ATE	-	Automatic Test Equipment
BIN	-	BINary
BIT	-	Built-In Test
CAD	-	Cartridge Activated Device
CAEA	-	Complete And Enter All
CAEO	-	Complete And Enter One
CAMS	-	Core Automated Maintenance System
CDM	-	Content Data Model
CLIPS	-	C Language Production System
CND	-	Can Not Duplicate
DC	-	Damage Code
EAFF	-	Exit At First Failure
FOD	-	Foreign Object Damage
HOW MAL	-	How Maifunction
I-SHOP	-	Intermediate Level Maintenance Shop
IMIS	-	Integrated Maintenance Information System
IMIS-DM	-	IMIS Diagnostic Module
LOC	-	Location
LRC	-	Combat Logistics Branch
LRU	-	Line Replaceable Unit
MDAS	-	Maintenance Diagnostic Aiding System
MOT	-	Multiple Outcome Test
MTBF	-	Mean Time Between Failure
NASA	-	National Aeronautics and Space Agency
NMC	-	Not Mission Capable
NMCS	-	Not Mission Capable/Supply
OO	-	Object Oriented
OPS	-	Operations
PMA	-	Portable Maintenance Aid
PMC	-	Partially Mission Capable
PROLOG	-	Programming in Logic
QPA	-	Quantity Per Application
RCM	-	Reliability Centered Maintenance
R&D	-	Research and Development
R&M	-	Reliability and Maintainability
R&R	-	Remove and Replace
SEI	-	Systems Exploration, Inc.
SRU	-	Shop Replaceable Unit
TO	-	Technical Order
TOC	-	Table Of Contents

GLOSSARY

Action. A diagnostic or corrective procedure performed by a maintenance technician.

Aircraft Configuration. Placements or layouts of aircraft system components.

Availability. A component's or test equipment's obtainability for use in the diagnostics process.

Best Rectification. A multiple fault algorithm that chooses the optimum action from among available rectification actions.

Best Test. A diagnostic software algorithm that chooses the optimum test from among those available at any point in the diagnostic sequence.

Component. The lowest physical level of indenture on which a maintenance technician at a given level of maintenance (i.e., organizational, intermediate, and depot (O, I, D)) will normally work. For example, an organizational-level maintenance technician would consider a Line Replaceable Unit (LRU) as a component, whereas an intermediate-level technician would consider the LRU an end item and the Shop Replaceable Unit (SRU) a component.

Criticality. A measure of need for a particular system capability. For example, a fault in an air-to-ground function might not be critical for an air defense sortie whereas a fault in an air-to-air function would be critical for the same sortie requirement.

Dominant Action. A rectification action whose likelihood of success is so great that it is recommended before available tests that would reduce the plausible set.

Failure Rate. The inverse of Mean Time Between Failure (MTBF).

Fault. The cause of an equipment malfunction. The manifestation, through either inference or direct observation, of a failure or failures within a system. Also used here to refer to any possible faults that can cause a symptom.

Feedback Analysis. The process of collecting parameters while in the maintenance/diagnostic environment and using these parameters to update current logistics information.

Feedback Loop. An interconnection of faults and signals such that no single test point can successfully isolate the fault location.

Functional Check. A test performed to ensure a rectification action has been successful in restoring a system to operational status.

Mean Time Between Failure (MTBF). The unit of reliability used as a predictor of fault likelihood. Its inverse is the failure rate.

Multiple Faults. An event where two or more faults (failed components) exist simultaneously in a given system.

Multiple Outcome Test (MOT). A test procedure without a binary pass/fail result. The procedure may have any number of outcomes; however, each outcome is unique and distinguishable from all other outcomes.

GLOSSARY (Concluded)

Plausible Set. The set of possible faults that could logically have led to an observed or indicated faulty condition. The elements in this set of faults contain single faults or combinations of faults that are not redundant.

Rectification. The repair of a fault(s) which alleviates a symptom or set of symptoms.

Repair Time. The time required to complete system repair after a fault is isolated. It may include access times. It will include reinstallation of original components removed unnecessarily as part of diagnostics, secure and closure, and final functional check.

Support Equipment. Tools or devices needed to perform an action.

Symptom. A machine-generated or verbal description code indicating that a malfunction exists (e.g., "Receiver, no audio").

Test. A prescribed sequence of actions whose result will implicate or exonerate a set of faults.

Test Time. The time required to perform a test. It includes access time, time to gather necessary test equipment and tools, time to conduct the test procedures, and time needed to record/interpret test results.

Time to Repair. The time required to perform a repair action or rectification.

APPENDIX A: SMALLTALK DEFINITIONS

SMALLTALK DEFINITIONS

Passing Information to The Diagnostic Module Controller

Critical Faults. Informing the diagnostic module controller of critical faults is done through the "setCriticalFaults:" message. This message requires a parameter consisting of a list of critical faults.

Observed Symptoms. To manually set observed symptoms the "addSymptoms:" message is used with a list of symptoms.

Results of a Test. The "ranTest:withOutcomes:" message is used to pass the results of a test. This message requires two parameters. The first parameter is the instance of test performed. The second is a list of outcomes observed as a result of performing the test.

Rectification Performed. The "performRectification:" message is used to inform the diagnostic controller a rectification has been performed. The parameter passed along with this message is the rectification.

Changed Results from a Previous Test. The message "change Results From Previous Test: with Outcomes:" is used to change results from a previous test. The first parameter used for this method is a test element from the previous actions list. The elements from the previous actions list contain necessary information. The necessary information includes the machine's state before the test was performed, as well as the test. The second parameter is the new outcomes or test results. The current machine state is replaced with the one stored in the previous action element. Now the machine state is modified with the new test results. Using other tasks performed after the test was run initially, the current machine state is updated.

Retrieving Information from the Diagnostic Module Controller

Obtaining the List of Previous Actions. To obtain a previous actions list from the diagnostic controller, use the "previousActions" method. This method returns an ordered list of tests and rectifications. The test items also contain the observed outcomes of the test.

Obtain the Best Task. To obtain the top interleaved task from the diagnostic controller, use the "bestTask" message.

Obtaining Ranked Lists. Three types of ranked lists are obtainable from the diagnostic controller: a) interleaved tasks, b) ranked tests, and c) ranked rects. The interleaved tasks list is a ranked listing of tests and rectifications based on the dominant action algorithm. To retrieve an interleaved tasks list from the diagnostic controller, use the "interleavedTasks" message. The ranked tests list is an ordered list of tests based on the best test algorithm. To obtain the ranked tests list from the diagnostic controller, use the "rankedTests" message. The ranked rectifications list is an ordered list of rectifications based on the best rects algorithm. To obtain the ranked rectifications list from the diagnostic controller, use the "rankedRects" message.

APPENDIX B: SMALLTALK/V IMIS-DM CLASS STUCTURE

SMALLTALK/V IMIS-DM CLASS STRUCTURE

Several classes were created to implement the Smalltalk version of the diagnostic module:

1. MDAScontroller
2. MDASpreviousAction
3. MDASdiagnostics
4. MDASdiagnosticsUsingCombinations (subclass of MDASdiagnostics)
5. MDASsets
6. MDAScombinationsGenerator
7. MDAScombination
8. MDAStasks
9. MDASrankedTasks
10. MDAS PhysDiagnostics (Physical model normal diagnostics interface)
11. PhysModel (Physical Model)

Figure A shows those classes used by the other MDAS classes to perform their functions and where those classes reside.

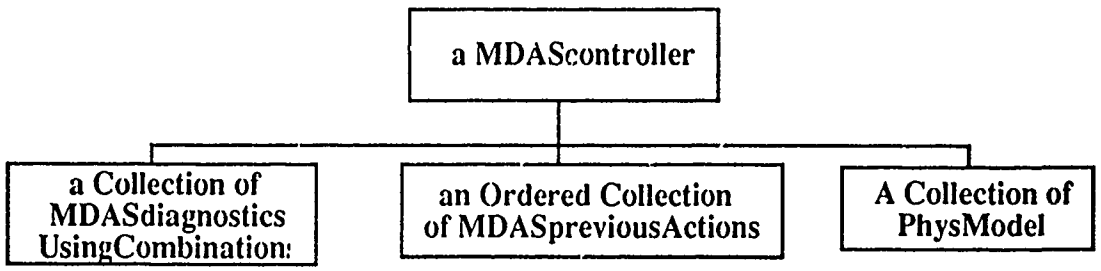


Figure A. IMIS-DM Usage.

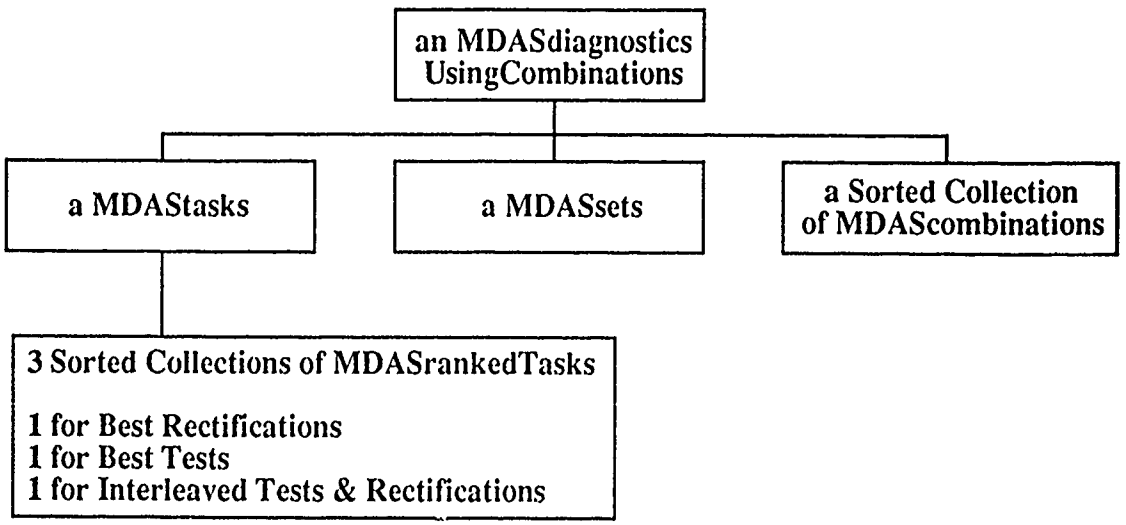


Figure B. IMIS-DM Functional Module Usage.

The MDAScombinationsGenerator class is not shown in Figure B. This class is used briefly by instances of MDASdiagnosticsUsingCombinations to generate a sorted collection of MDAScombinations. The MDAScontroller class is the public interface into the diagnostic system. One instance of the MDAScontroller is needed to perform diagnostics. This instance creates, monitors, and controls the creation and deletion of other instances of the MDAS classes. The following is a description of each of the MDAS classes. The descriptions will include a class description and class methods with a description and instance methods with a description. The date, usage, and type of interface (Public/Private) are also given.

Class: MDAScombination

Description:

This class describes the fault combinations used in diagnostics.

Instance Variables:

combinationFaults - A list of the faults in the combination.
combinationProbability - The probability of the combination.

Class Methods:

new

Date: Feb 6, 1990
Usage: new
Interface: Public

Create a new instance of MDAScombination.

Instance Methods:

addFault:withProbability:

Date: Feb 6, 1990
Usage: addFault: aFault withProbability: aFaultProbability
Interface: Public

Add a fault to the combination and update the combination probability with the fault's probability.

combinationFaults

Date: Feb 6, 1990
Usage: combinationFaults
Interface: Public

Answer the collection of faults that make up the combination.

combinationProbability

Date: Feb 6, 1990
Usage: combinationProbability
Interface: Public

Answer the probability of the combination.

combinationProbability:

Date: Feb 6, 1990
Usage: combinationProbability: aValue
Interface: Public

Set the probability of the combination.

contains:

Date: Feb 6, 1990
Usage: contains: aFault
Interface: Public

See if a fault is contained in a combination.

containsAny:

Date: Feb 6, 1990
Usage: containsAny: aCollection
Interface: Public

See if any faults in a collection are contained in a combination.

initialize

Date: Feb 6, 1990
Usage: initialize
Interface: Public

Initialize a new combination.

printOn:

Date: Feb 6, 1990
Usage: printOn: aStream
Interface: Public

Modify the standard printOn: meassage.

Class: MDAScombinationsGenerator

Description:

This class will generate a list of fault combinations using the observed symptoms.

Instance Variables:

symptoms - A list of the observed symptoms.
faults - A list of maybe set faults to combine.
symptomFaultMatrix - A matrix which holds for each symptom the probabilities that each of the faults caused the symptom.
faultProbabilities - A list of probabilities for each of the faults. These probabilities will be an average.
combinations - A list of generated combinations.
combinedFaults - Used when finding the faults in a combination.
remainingFaults - The faults to be searched for the current combination.
remainingSymptoms - The symptoms that still need to be searched for faults to combine.

Class Methods:

new

Date: Feb 6, 1990
Usage: new
Interface: Public

Create a new instance of MDAScombinationsGenerator.

Instance Methods:

addCombination

Date: Feb 5, 1990
Usage: addCombination
Interface: Private

Create a combination from the collection of combined faults, and add it to the collection of combinations.

buildCombinations

Date: Feb 5, 1990
Usage: buildCombinations
Interface: Private

Search for faults in a combination. If there are more symptoms, then recursively call buildCombinations. When there are no more symptoms, a complete combination has been created; so, store the combination in combinations.

combinations

Date: Feb 5, 1990
Usage: combinations
Interface: Public

Answer the list of combinations.

faultProbabilities

Date: Jan 17, 1990
Usage: faultProbabilities
Interface: Public

Answer the dictionary of fault probabilities.

faults:

Date: Feb 5, 1990
Usage: faults: aFaultList
Interface: Public

Set the list of faults that need to be combined.

getFaultProbabilities

Date: Feb 5, 1990
Usage: getFaultProbabilities
Interface: Private

Calculate the fault probabilities by taking the average probabilities of each fault for all symptoms.

getFaultsAndWeights:

Date: Feb 5, 1990
Usage: getFaultsAndWeights: aSymptom
Interface: Private

Answer a dictionary with faults as keys and weights as values for a symptom.

getNextFault

Date: Feb 5, 1990
Usage: getNextFault
Interface: Private

Answer the next fault to find combinations for.

getNextFaultUsing:

Date: Feb 5, 1990
Usage: getNextFaultUsing: possibleFaults
Interface: Private

Answer the next fault to find combinations for using the possible faults.

getPossibleFaults

Date: Feb 5, 1990
Usage: getPossibleFaults
Interface: Private

Answer a list of possible faults given the remaining symptoms and remaining faults.

initialize

Date: Feb 5, 1990
Usage: initialize
Interface: Public

Initialize the needed instance variables.

isFault:inSymptom:

Date: Feb 5, 1990
Usage: isFault: aFault inSymptom: aSymptom
Interface: Private

Answer true if a fault is in a matrix row.

makeCombinations

Date: Feb 5, 1990
Usage: makeCombinations
Interface: Public

Create a list of combinations.

makeSymptomFaultMatrix

Date: Feb 5, 1990
Usage: makeSymptomFaultMatrix
Interface: Private

Create a symptom/fault matrix.

removeSymptomsThatImplicate:

Date: Feb 5, 1990
Usage: removeSymptomsThatImplicate: aFault
Interface: Private

Remove the symptoms in remaining symptoms that implicate the fault.
Answer the collection of removed symptoms.

replaceSymptoms:

Date: Feb 5, 1990
Usage: replaceSymptoms: removedSymptoms
Interface: Private

Replace the set of removed symptoms in set of remaining symptoms.

symptoms

Date: Feb 5, 1990
Usage: symptoms
Interface: Public

Answer the list of symptoms that need to be searched for combinations.

symptoms:

Date: Feb 5, 1990
Usage: symptoms: aSymptomList
Interface: Public

Set the list of symptoms that need to be searched for combinations.

traverseMatrix

Date: Feb 5, 1990
Usage: traverseMatrix
Interface: Private

Traverse the symptom/fault matrix to create a list of combinations.

Class: MDAScontroller

Description:

This class will control the creation and status of MDASdiagnostics.

Instance Variables:

diagnosticGroups - A list of independent diagnostic groups.
currentDiagnostic - The diagnostic group being worked on.
criticalFaults - A list of critical faults.
previousActions - A list of tasks that were performed.
objectBase - The object base where diagnostic information is held.
testClass - The key for accessing all tests from the object base.
rectClass - The key for accessing all rectifications from the object base.
exculpatedFaults - A list of the exculpated faults from fixed diagnostic groups.
tocRects - A list of all the rectifications.
tocTests - A list of all the tests.

Class Methods:

new

Date: Jan 16, 1990
Usage: new
Interface: Public

Create a new instance of MDAScontroller.

Instance Methods:

degradedbestTask

Date: Mar 19, 1990
Usage: degradedbestTask
Interface: Public

Answer the top task from smart TOC in degraded mode.

degradedinterleavedTasks

Date: Mar 19, 1990
Usage: degradedinterleavedTasks
Interface: Private

Answer a list of Tests for degraded mode.

degradedMode

Date: Aug 3, 1990
Usage: degradedMode
Interface: Public

Set myself in degraded mode.

degradedrankedRects

Date: Mar 19, 1990
Usage: degradedrankedRects
Interface: Private

Answer a list of ranked rects for degraded mode.

degradedrankedTests

Date: Mar 19, 1990
Usage: degradedrankedTests
Interface: Private

Answer a list of ranked tests for degraded mode.

doesNotUnderstand:

Date: Jul 24, 1990
Usage: doesNotUnderstand: aMessage
Interface: Public

Self does not understand a message; so, it will pass it to the appropriate model. The functional model is default. If the functional model does not understand the message, an error is generated.

functionalModel

No documentation available.

functionalModel:

No documentation available.

initialize

Date: Jul 24, 1990
Usage: initialize
Interface: Public

Initialize the needed instance variables.

jumpToFunctionalMode

Date: Jul 30, 1990
Usage: jumpToFunctionalMode
Interface: Private

When a functional test fails, jump to normal diagnostics until system okay.

mode

Date: Jul 24, 1990
Usage: mode
Interface: Public

Answer the mode of operation (i.e., degraded, normal, physical).

mode:

Date: Jul 30, 1990
Usage: mode: aModeOfOperation
Interface: Public

Set the mode of operation (i.e., degraded, normal, physical).

normalMode

Date: Aug 3, 1990
Usage: normalMode

Interface: Public

Set myself in normal mode.

objectBase

Date: Mar 21, 1990
Usage: objectBase
Interface: Public

Answer the object base.

objectBase:

Date: Jul 26, 1990
Usage: objectBase: anObjectBase
Interface: Public

Set the object base and the element mapping.

passToDegradedModel:

Date: Jul 25, 1990
Usage: passToDegradedModel: aMessage
Interface: Public

Self does not understand aMessage and is in degraded mode; therefore, check to see if the degraded model understands it. If not, try the functional model by default.

passToFunctionalModel:

Date: Jul 25, 1990
Usage: passToFunctionalModel: aMessage
Interface: Public

Self does not understand aMessage and is passing it to functional model by default.

passToPhysicalModel:

Date: Jul 25, 1990
Usage: passToPhysicalModel: aMessage
Interface: Public

Self does not understand aMessage and is in physical mode; therefore, check to see if the physical model understands it. If not, try the functional model by default.

performRectification:

Date: Jul 30, 1990
Usage: performRectification: aRect
Interface: Public

Pass the performed rectification to the functional model and the physical model if appropriate.

physicalHold

No documentation available.

physicalHold:

No documentation available.

physicalMode

Date: Aug 3, 1990
Usage: physicalMode
Interface: Public

Set self in physical mode.

physicalModel

Date: Jul 24, 1990
Usage: physicalModel
Interface: Public

Answer the physical model.

physicalModel:

Date: Jul 24, 1990
Usage: physicalModel: aPhysModel
Interface: Public

Set the physical model.

ranTest:withOutcomes:

Date: Jul 30, 1990
Usage: ranTest: aTest withOutcomes: outcomes
Interface: Public

Pass the performed test with observed outcomes to the appropriate functional model, then perform necessary actions if in physical mode.

returnToPhysicalMode

Date: Jul 30, 1990
Usage: returnToPhysicalMode
Interface: Private

When a functional test passes, jump back to physical diagnostics.

startPhysicalAssessment

Date: Aug 7, 1990
Usage: startPhysicalAssessment

Interface: Public

Initialize and start the physical model.

tocRects

Date: Mar 20, 1990
Usage: tocRects
Interface: Private

For degraded mode answer the current set of TOC rectifications.

tocTests

Date: Mar 20, 1990
Usage: tocTests
Interface: Private

For degraded mode answer the current set of TOC tests.

Class: MDASdiagnostics

Description:

This class stores and manipulates the state of a diagnostic group.

Instance Variables:

diagnosticSets - Holds on to a class which keeps track of the state of each fault.
faultProbabilites - The average probability for each possible fault.
combinations - A list of fault combinations.
observedSymptoms - A list of observed symptoms.
tasks - Holds on to a class that keeps the tasks (test & rects).
doSecondStep - True or False depending if second step is to be performed.
failedExculpated - Keeps track of the number of times a test fails only those faults that have been exculpated.
usedRectified - Keeps track of the number of times the rectified faults have been returned to the maybe set.
objectBase - The object base where diagnostic information is held.
testClass - The key for accessing all tests from the object base.

Class Methods:

new

Date: Mar 21, 1990
Usage: new
Interface: Public

Create a new instance of MDASdiagnostics.

Instance Methods:

addImplicatedSymptomsFrom:withOutcomes:

Date: Feb 5, 1990
Usage: addImplicatedSymptomsFrom: aTest withOutcomes: outcomes
Interface: Private

Add the appropriate outcomes of a test to the observed symptoms.

addRects:

Date: Feb 5, 1990
Usage: addRects: aRectsList
Interface: Public

Add a list of rectifications to the rects set.

addSymptom:

Date: Feb 5, 1990
Usage: addSymptom: aSymptom
Interface: Public

Add a symptom to the observed symptoms list.

addSymptoms:

Date: Feb 5, 1990
Usage: addSymptoms: aSymptomList
Interface: Public

Add symptoms from a symptom list to the observed symptoms.

addTests:

Date: Feb 5, 1990
Usage: addTests: aTestsList
Interface: Testing

Add a list of tests to the tests set.

adjustTestValue:with:

Date: Feb 5, 1990
Usage: adjustTestValue: aRankedTest with: maybeSetProbability
Interface: Private

Adjust the test coefficients for the probability of failing outside the plausible but in the maybe set.

allExculpated:

Date: Feb 5, 1990
Usage: allExculpated: aFaultList
Interface: Private

Answer true if all the faults in a fault list are exculpated.

allExculpatedOrRectified:

Date: Feb 5, 1990
Usage: allExculpatedOrRectified: aFaultList
Interface: Private

Answer true if all the faults in a fault list are exculpated or rectified.

allUnion:

Date: Feb 5, 1990
Usage: allUnion: aFaultList
Interface: Private

Answer true if all the faults in a fault list are in the union.

allUnionExculpatedOrRectified:

Date: Feb 5, 1990
Usage: allUnionExculpatedOrRectified: aFaultList
Interface: Private

Answer true if all the faults in a fault list that are in the union are exculpated or rectified. If no faults in the union, answer false.

buildPlausibleSet

Date: Feb 5, 1990
Usage: buildPlausibleSet
Interface: Private

Build the plausible set from the faults in the union that are contained in any combination.

checkIfCriticalRect:

Date: May 22, 1990
Usage: checkIfCriticalRect: aFaultList
Interface: Private

Set the criticality status for the spanned faults of a rectification.

checkIfCriticalTest:

Date: May 22, 1990
Usage: checkIfCriticalTest: aFaultList

Interface: Private

Set the criticality status for the spanned faults of a test.

compareCriticalityAndBestTest:

Date: May 25, 1990
Usage: compareCriticalityAndBestTest: tempRankedTests
Interface: Private

Compare the top critical test to the best test.

containsCriticals:

Date: May 25, 1990
Usage: containsCriticals: aFaultList
Interface: Private

Answer true if any of the faults in a fault list are critical.

criticalFaults

Date: May 16, 1990
Usage: criticalFaults
Interface: Private

Answer the critical faults.

criticalitySet

Date: Feb 13, 1990
Usage: criticalitySet
Interface: Private

Answer true if there are any critical faults set; otherwise, answer false.

criticalsInList:

Date: May 22, 1990
Usage: criticalsInList: aList
Interface: Private

Answer the critical faults in a list.

deepCopy

Date: Feb 5, 1990
Usage: deepCopy
Interface: Public

Answer a copy of the receiver with deep copies of each instance variable.

diagnosticSets

Date: Feb 5, 1990
Usage: diagnosticSets
Interface: Private

Answer the diagnostic sets class.

does:span:

Date: Feb 5, 1990
Usage: does: spannedFaults span: aFaultList
Interface: Private

See if any of the spanned faults are in a fault list.

doesSplitPlausible:

Date: Apr 30, 1990
Usage: doesSplitPlausible: spannedFaults
Interface: Private

See if the spanned faults split the plausible faults.

doSecondStep

Date: Feb 5, 1990
Usage: doSecondStep
Interface: Private

Answer whether or not second step is to be performed.

doSecondStep:

Date: Feb 5, 1990
Usage: doSecondStep: aBoolean
Interface: Private

Set do Second Step to true or false. If true then interleaving tasks will perform second step calculations; interleaving tasks will not perform second step if false.

enhancementFactor:

Date: Mar 8, 1990
Usage: enhancementFactor: tempRankedTests
Interface: Private

Add the enhancement factor for access groups for tests.

evaluateTests:

Date: May 22, 1990
Usage: evaluateTests: tempRankedTests
Interface: Private

Get the coefficients for the tests.

exculpatedFaults

Date: Feb 5, 1990
Usage: exculpatedFaults
Interface: Public

Answer the exculpated faults.

failedExculpated

Date: Mar 16, 1990
Usage: failedExculpated
Interface: Private

Answer the number of times a test fails faults that have been exculpated.

faultProbabilities

Date: Feb 5, 1990
Usage: faultProbabilities
Interface: Private

Answer the dictionary of fault probabilities.

faultProbabilities:

Date: Feb 5, 1990
Usage: faultProbabilities: aFaultProbabilityList
Interface: Private

Set the dictionary of fault probabilities.

faultProbabilityMultipliers

Date: Aug 7, 1990
Usage: faultProbabilityMultipliers
Interface: Public

Answer the fault probability multipliers dictionary.

faultProbabilityMultipliers:

Date: Aug 7, 1990
Usage: faultProbabilityMultipliers: aDictionary
Interface: Public

Set the fault probability multipliers dictionary.

getProbabilities

Date: Apr 30, 1990

Usage: getProbabilities
Interface: Private

Get the probabilities from the fault probabilities list.

getRectsFailureProbabilities:

Date: May 22, 1990
Usage: getRectsFailureProbabilities: tempRankedRects
Interface: Private

Get the probability of failure for the rects.

getRectsFrom:

Date: Feb 5, 1990
Usage: getRectsFrom: aFaultList
Interface: Public

Answer a collection of rects from a fault list.

getTestsFrom:

Date: Feb 5, 1990
Usage: getTestsFrom: aFaultList
Interface: Public

Answer a collection of tests from a fault list.

initialize

Date: Feb 5, 1990
Usage: initialize
Interface: Public

Initialize the instance variables.

initializePlausibleSet

Date: Feb 5, 1990
Usage: initializePlausibleSet
Interface: Public

Initialize the plausible set and rank tasks.

interleavedTasks

Date: Feb 5, 1990
Usage: interleavedTasks
Interface: Public

Answer a list of interleaved Tests and Rects.

interleaveTasks

Date: Feb 5, 1990
Usage: interleaveTasks
Interface: Private

Interleave the tests and rects.

isolatedFaults

Date: Feb 7, 1990
Usage: isolatedFaults
Interface: Public

Answer the isolated faults list.

maybeSetFaults

Date: Feb 5, 1990
Usage: maybeSetFaults
Interface: Public

Answer the maybe set faults.

modifyDiagnosticSetsFrom:and:

Date: Feb 7, 1990
Usage: modifyDiagnosticSetsFrom: exculpatedFaults and: implicatedFaults
Interface: Public

Use the exculpated and implicated faults to modify the state of the system.

modifyProbabilitiesOf:with:

Date: Aug 7, 1990
Usage: modifyProbabilitiesOf: aFaultList with: aMultiplier
Interface: Public

Place every fault in a list in the fault probability multipliers along with a multiplier.

modifySymptomsUsing:withOutcomes:

Date: Feb 7, 1990
Usage: modifySymptomsUsing: aTest withOutcomes: outcomes
Interface: Public

Remove or add symptoms based on the state of the faults and the results of a test with the outcomes.

moveMaybeToPlausible

Date: Apr 30, 1990
Usage: moveMaybeToPlausible

Interface: Private

Move the faults that are in maybe set to the plausible set.

normalizeProbabilities

Date: Feb 5, 1990
Usage: normalizeProbabilities
Interface: Private

Normalize the combination probabilities.

noUnion:

Date: Feb 5, 1990
Usage: noUnion: aFaultList
Interface: Private

Answer true if none of the faults in a fault list are in the union.

numberOfCriticalFaults

Date: Feb 5, 1990
Usage: numberOfCriticalFaults
Interface: Public

Answer the number of critical faults.

numberOfPlausibles:

Date: Feb 5, 1990
Usage: numberOfPlausibles: aList
Interface: Private

Answer the number of plausible faults in a list.

numberOfTotalPlausibles

Date: Apr 30, 1990
Usage: numberOfTotalPlausibles
Interface: Private

Answer the number of plausible faults.

objectBase

Date: Feb 15, 1990
Usage: objectBase
Interface: Public

Answer the object base.

objectBase:

Date: Feb 15, 1990
Usage: objectBase: anObjectBase
Interface: Public

Set the object base.

observedSymptoms

Date: Feb 5, 1990
Usage: observedSymptoms
Interface: Private

Answer the list of observedSymptoms.

performRectification:

Date: Feb 5, 1990
Usage: performRectification: aRect
Interface: Public

Perform a rectification.

plausibleFaults

Date: Feb 5, 1990
Usage: plausibleFaults
Interface: Public

Answer the plausible faults.

plausiblesInList:

Date: Feb 5, 1990
Usage: plausiblesInList: aList
Interface: Private

Answer the plausible faults in a list.

plausiblesLessList:

Date: May 15, 1990
Usage: plausiblesLessList: aList
Interface: Private

Answer the plausible faults less the faults in a list.

probOfList:

Date: May 22, 1990
Usage: probOfList: aFaultList
Interface: Private

Answer the sum of probabilities of the plausible faults in a list.

rankedRects

Date: Feb 5, 1990
Usage: rankedRects
Interface: Public

Answer a list of ranked rects.

rankedTests

Date: Feb 5, 1990
Usage: rankedTests
Interface: Public

Answer a list of ranked tests.

rankRects

Date: Feb 5, 1990
Usage: rankRects
Interface: Private

Rank the rects.

rankSecondStep

Date: Jan 18, 1990
Usage: rankSecondStep
Interface: Private

Rank the tests, rects, and interleaving for second step. Set doSecondStep to false.

rankTasks

Date: Feb 5, 1990
Usage: rankTasks
Interface: Public

Rank the tests, rects, and interleaving. Set doSecondStep to true.

rankTests

Date: May 15, 1990
Usage: rankTests
Interface: Private

Rank the tests.

ranNormalTest:withOutcomes:exculpated:implicated:

Date: Feb 5, 1990

Usage: ranNormalTest: aTest withOutcomes: outcomes exculpated: tempExculpated implicated: tempFailed
Interface: Private

Get the exculpated and implicated faults from the normal test that observed these outcomes.

ranQAFFTest:withOutcomes:exculpated:implicated:

Date: Feb 5, 1990
Usage: ranQAFFTest: aTest withOutcomes: outcomes exculpated: tempExculpated implicated: tempFailed
Interface: Private

Get the exculpated and implicated faults from the Quit At First Failure (QAFF) test that observed these outcomes.

ranTest:withOutcomes:

Date: Feb 5, 1990
Usage: ranTest: aTest withOutcomes: outcomes
Interface: Public

Ran the test and observed these outcomes. Pass the results to the appropriate test type and get the exculpated and implicated faults.

rectifiedFaults

Date: Feb 5, 1990
Usage: rectifiedFaults
Interface: Public

Answer the rectified faults.

rects

Date: Feb 5, 1990
Usage: rects
Interface: Public

Answer a collection of rectifications.

removeExculpatedSymptoms

Date: Feb 5, 1990
Usage: removeExculpatedSymptoms
Interface: Private

Remove all symptoms that are entirely exculpated.

removeFromConsideration:

Date: Apr 30, 1990
Usage: removeFromConsideration: aFaultList

Interface: Private

Remove the faults in the fault list from the fault probabilities list.

removeSymptom:

Date: Feb 5, 1990
Usage: removeSymptom: aSymptom
Interface: Public

Remove a symptom from the observed symptoms list.

removeSymptoms:

Date: Feb 5, 1990
Usage: removeSymptoms: aSymptomList
Interface: Public

Remove symptoms in the list from the observed symptoms list.

reRunTest:withOutcomes:

Date: Feb 5, 1990
Usage: reRunTest: aTest withOutcomes: outcomes
Interface: Private

For adding new symptoms, rerun the test with the observed outcomes. Pass the results to the appropriate test type and get the exculpated and implicated faults.

secondStep:with:

Date: Feb 5, 1990
Usage: secondStep: aRankedRect with: aRankedTest
Interface: Private

Answer true if the rect is better than the test after second step. Otherwise, answer false.

secondStepRectCost:

Date: Feb 5, 1990
Usage: secondStepRectCost: aRankedRect
Interface: Private

Answer the rect cost for second step.

secondStepTestCost:

Date: May 22, 1990
Usage: secondStepTestCost: aRankedTest
Interface: Private

Answer the test cost for second step.

setCriticalFaults:

Date: Feb 5, 1990
Usage: setCriticalFaults: criticalFaultsList
Interface: Public

Set the critical faults that are plausible and answer the number of critical faults.

setMaybeToUnionLessExculpated

Date: Feb 5, 1990
Usage: setMaybeToUnionLessExculpated
Interface: Private

Make the maybe set to the union set less the exculpated set.

setMaybeToUnionLessExculpatedAndRectified

Date: Feb 5, 1990
Usage: setMaybeToUnionLessExculpatedAndRectified
Interface: Private

Make the maybe set to the union set less the exculpated and rectified sets.

sumProbOfPlausible

Date: May 25, 1990
Usage: sumProbOfPlausible
Interface: Private

Answer the sum of probabilities of the plausible faults.

symptomsThatImplicate:

Date: Feb 7, 1990
Usage: symptomsThatImplicate: aFault
Interface: Public

Answer only the symptoms that implicate a fault.

symptomsToBeUsed

Date: Feb 7, 1990
Usage: symptomsToBeUsed
Interface: Public

Answer only the symptoms that are to be used to calculate the plausible set.

tasks

Date: Feb 5, 1990
Usage: tasks
Interface: Private

Answer the instance of the tasks class.

tests

Date: Feb 5, 1990
Usage: tests
Interface: Public

Answer a collection of tests.

testValueAdjustment:

Date: Feb 5, 1990
Usage: testValueAdjustment: tempRankedTests
Interface: Private

Adjust the test coefficients for the probability of failing outside the plausible but in the maybe set.

totalStepsFor:

Date: May 25, 1990
Usage: totalStepsFor: aTest
Interface: Private

Calculate the total number of steps to repair if a test is performed.

totalStepsToFlyFor:

Date: May 25, 1990
Usage: totalStepsToFlyFor: aCriticalTest
Interface: Private

Calculate the total number of steps to repair if a critical test is performed.

unionFaults

Date: Feb 5, 1990
Usage: unionFaults
Interface: Public

Answer the union faults list.

updateUsefulRects

Date: Feb 5, 1990
Usage: updateUsefulRects
Interface: Private

Remove all unnecessary rectifications from useful rects set.

updateUsefulTests

Date: Feb 5, 1990

Usage: updateUsefulTests
Interface: Private

Remove all unnecessary tests from useful tests set.

usedRectified

Date: Mar 16, 1990
Usage: usedRectified
Interface: Private

Answer the number of times the rectified set was used to build the plausible set.

Class: MDASdiagnosticsUsingCombinations

Instance Methods:

combinations

Date: Apr 30, 1990
Usage: combinations
Interface: Private

Answer the list of combinations.

combinations:

Date: Apr 30, 1990
Usage: combinations: aCombinationList
Interface: Private

Set the list of combinations.

doesSplitPlausible:

Date: Apr 30, 1990
Usage: doesSplitPlausible: spannedFaults
Interface: Private

See if the spanned faults split the plausible faults and the combinations.

getProbabilities

Date: Apr 30, 1990
Usage: getProbabilities
Interface: Private

Get a list of combinations from the combination generator class.

initialize

Date: Apr 30, 1990

Usage: initialize
Interface: Public

Initialize the instance variables.

isStillCombined:

Date: Apr 30, 1990
Usage: isStillCombined: aFault
Interface: Private

Answer true if a fault is still in any combinations; otherwise, answer false.

moveMaybeToPlausible

Date: Apr 30, 1990
Usage: moveMaybeToPlausible
Interface: Private

Move the faults that are in any of the combinations from the maybe set to the plausible set.

normalizeProbabilities

Date: Apr 30, 1990
Usage: normalizeProbabilities
Interface: Private

Normalize the combination probabilities.

numberOfCombinations

Date: Apr 30, 1990
Usage: numberOfCombinations
Interface: Private

Answer the number of combinations.

numberOfPlausibles:

Date: Apr 30, 1990
Usage: numberOfPlausibles: aList
Interface: Private

Answer the number of combinations spanned by the plausible faults in a list.

numberOfTotalPlausibles

Date: Apr 30, 1990
Usage: numberOfTotalPlausibles
Interface: Private

Answer the number of combinations.

probOfList:

Date: May 22, 1990
Usage: probOfList: aFaultList
Interface: Private

Answer the sum of probabilities of combinations that contain the plausible faults in a list.

removeFromConsideration:

Date: Apr 30, 1990
Usage: removeFromConsideration: aFaultList
Interface: Private

Remove the combinations that contain any of the faults in the fault list. Afterwards, check each of the faults that combined with the faults in the list to see if they are in any other combinations; if not, place them in the maybe set.

Class: MDASfunctionalController

Description:

This class will control the creation and status of MDASdiagnostics.

Instance Variables:

diagnosticGroups - A list of independent diagnostic groups.
currentDiagnostic - The diagnostic group being worked on.
criticalFaults - A list of critical faults.
previousActions - A list of tasks that were performed.
objectBase - The object base where diagnostic information is held.
testClass - The key for accessing all tests from the object base.
rectClass - The key for accessing all rectifications from the object base.
exculpatedFaults - A list of the exculpated faults from fixed diagnostic groups.
tocRects - A list of all the rectifications.
tocTests - A list of all the tests.

Class Methods:

new

Date: Jan 16, 1990
Usage: new
Interface: Public

Create a new instance of MDASfunctionalController.

Instance Methods:

addSymptom:

Date: Feb 6, 1990
Usage: addSymptom: aSymptom
Interface: Public

Find the appropriate diagno... group for a symptom and add the symptom to the group.

addSymptoms:

Date: Feb 6, 1990
Usage: addSymptoms: anObservedSymptomsList
Interface: Public

Find the appropriate diagnostic groups for the symptoms and add symptoms to their respective group.

addToCritical:

Date: Feb 6, 1990
Usage: addToCritical: aFault
Interface: Public

Add a fault to the critical set.

addToPreviousActions:

Date: Feb 6, 1990
Usage: addToPreviousActions: aPreviousAction
Interface: Private

Add a previous action to the ordered list of previous actions.

aGroup:has:or:

Date: Feb 6, 1990
Usage: aGroup: aDiagnosticGroup has: rects or: tests
Interface: Private

Answer true if the Group has any of the rects or tests.

aGroup:hasRect:

Date: Feb 6, 1990
Usage: aGroup: aDiagnosticGroup hasRect: aRect
Interface: Private

Answer true if the Group has a rect.

aGroup:hasTest:

Date: Feb 6, 1990
Usage: aGroup: aDiagnosticGroup hasTest: aTest
Interface: Private

Answer true if the Group has a test.

bestTask

Date: Jul 24, 1990
Usage: bestTask
Interface: Public

Answer the top interleaved task in normal mode.

changeResultsFromPreviousTest:withOutcomes:

Date: Feb 6, 1990
Usage: changeResultsFromPreviousTest: aPreviousTest withOutcomes: outcomes
Interface: Public

Change the results of a previous test. Start with the diagnostic state before the test was run. Then search for all other previous actions in the same diagnostic group as the test. When these actions are found, rerun them with the new diagnostic state.

checkCriticalityState

Date: Feb 6, 1990
Usage: checkCriticalityState
Interface: Private

Answer true if there are criticals and all are exculpated.

criticalFaults

Date: Feb 6, 1990
Usage: criticalFaults
Interface: Private

Answer the list of critical faults.

currentDiagnostic

Date: Feb 6, 1990
Usage: currentDiagnostic
Interface: Private

Answer the current diagnostic set being worked on.

currentDiagnostic:

Date: Feb 6, 1990
Usage: currentDiagnostic: aDiagnostic
Interface: Private

Set the current diagnostic set to be worked on.

diagnosticGroup:add:and:and:

Date: Feb 6, 1990
Usage: diagnosticGroup: diagnosticGroup add: aSymptom and: rects and: tests
Interface: Private

Add a symptom, rects and tests to a diagnostic group.

diagnosticGroups

Date: Feb 6, 1990
Usage: diagnosticGroups
Interface: Private

Answer the list of diagnostic groups.

diagnosticModeStatus

No documentation available.

diagnosticsClass

Date: May 2, 1990
Usage: diagnosticsClass
Interface: Private

Answer the class to be used in diagnostics.

exculpatedFaults

Date: Feb 6, 1990
Usage: exculpatedFaults
Interface: Public

Answer a list of all the exculpated faults from the diagnostic groups and the exculpated faults list.

faultNotFound

Date: Feb 6, 1990
Usage: faultNotFound
Interface: Private

Re-rank the tasks of the current diagnostic group.

getDiagnosticGroupFor:

Date: Feb 6, 1990
Usage: getDiagnosticGroupFor: aSymptom
Interface: Private

Get the group for a symptom. If no group present, add one.

getGroupWithRect:

Date: Feb 6, 1990
Usage: getGroupWithRect: aRect
Interface: Private

Answer the appropriate diagnostic group that has the rect.

getGroupWithTest:

Date: Feb 6, 1990
Usage: getGroupWithTest: aTest
Interface: Private

Answer the appropriate diagnostic group that has the test.

getRectsFrom:

Date: Feb 6, 1990
Usage: getRectsFrom: aFaultList
Interface: Private

Answer a collection of rects from a fault list.

getTestsFrom:

Date: Feb 6, 1990
Usage: getTestsFrom: aFaultList
Interface: Private

Answer a collection of tests from a fault list.

groupSymptoms:

Date: Feb 6, 1990
Usage: groupSymptoms: anObservedSymptomsList
Interface: Private

Group the symptoms into their perspective diagnostic groups.

initialize

Date: Feb 6, 1990
Usage: initialize
Interface: Public

Initialize the needed instance variables.

initializePlausibleSets

Date: Feb 6, 1990
Usage: initializePlausibleSets
Interface: Private

Initialize the plausible sets for all the diagnostic groups marked '#notInitialized'.

interleavedTasks

Date: Jul 24, 1990
Usage: interleavedTasks
Interface: Private

Answer a list of interleaved tests and rects for normal mode.

makeDiagnosticGroupFor:

Date: Feb 6, 1990
Usage: makeDiagnosticGroupFor: aFaultList
Interface: Private

Create a diagnostic group for a fault list. Add the tests and rects for the implicated faults to the diagnostic group. Answer the diagnostic group if a fault list is not empty; otherwise, answer nil.

maybeSetFaults

Date: Feb 6, 1990
Usage: maybeSetFaults
Interface: Public

Answer a list of all the maybe set faults from the diagnostic groups.

modifyRectProbabilities:

Date: Aug 7, 1990
Usage: modifyRectProbabilities: aProbabilityMultiplierDictionary
Interface: Public

Use the physical information to modify the weights of the faults spanned by a rect.

next

Date: Jul 26,1990
Usage: next
Interface: Public

Answer what method the presentation system is to perform next.

numberOfCriticalFaults

Date: Feb 6, 1990
Usage: numberOfCriticalFaults
Interface: Private

Answer the number of critical faults.

objectBase

Date: Mar 21, 1990
Usage: objectBase
Interface: Public

Answer the object base.

objectBase:

Date: Mar 21, 1990
Usage: objectBase: anObjectBase
Interface: Public

Set the object base.

performRectification:

Date: Feb 6, 1990
Usage: performRectification: aRect
Interface: Public

Pass the performed rectification to the appropriate diagnostic group.

plausibleFaults

Date: Feb 6, 1990
Usage: plausibleFaults
Interface: Public

Answer a list of all the plausible faults from the diagnostic groups.

previousActions

Date: Feb 6, 1990
Usage: previousActions
Interface: Public

Answer the ordered list of prevous actions.

rankedRects

Date: Jul 24, 1990
Usage: rankedRects
Interface: Private

Answer a list of ranked rects for normal mode.

rankedTests

Date: Jul 24, 1990
Usage: rankedTests
Interface: Private

Answer a list of ranked tests for normal mode.

ranTest:withOutcomes:

Date: Feb 6, 1990
Usage: ranTest: aTest withOutcomes: outcomes
Interface: Public

Pass the performed test with observed outcomes to the appropriate diagnostic group.

rectifiedFaults

Date: Feb 6, 1990
Usage: rectifiedFaults
Interface: Public

Answer a list of all the rectified faults from the diagnostic groups.

setCriticalFaults:

Date: Feb 6, 1990
Usage: setCriticalFaults: criticalFaultsList
Interface: Public

Set the critical faults list.

storeRect:for:

Date: Feb 6, 1990
Usage: storeRect: aRect for: diagnosticGroup
Interface: Private

Place the rect on the list of previous actions.

storeTest:withOutcomes:for:

Date: Feb 6, 1990
Usage: storeTest: aTest withOutcomes: outcomes for: diagnosticGroup
Interface: Private

Place the test and observed outcomes on the list of previous actions.

systemOK

Date: Feb 6, 1990

Usage: systemOK
Interface: Private

Answer '#systemOK' to inform that the system is OK.

whatsNext

Date: Jul 26, 1990
Usage: whatsNext
Interface: Private

Answer the thing to do next.

Class: MDASobjectBase

Description:

This class will be the interface between MDAS and its needed objects from the object base.

Class Methods:

new

Date: Jul 26, 1990
Usage: new
Interface: Public

Create a new instance of MDASobject Base.

Instance Methods:

allOcurrencesOf:

Date: Jul 26, 1990
Usage: allOcurrencesOf: anElement
Interface: Public

Answer all occurrences of an element from the external object base.

doesNotUnderstand:

Date: Jul 24, 1990
Usage: doesNotUnderstand: aMessage
Interface: Public

Pass object base the message if it understands it. Otherwise, initiate a walkback because a message was sent which is not understood; i.e., there is no matching method.

externalObjectBaseMap

Date: Jul 26, 1990
Usage: externalObjectBaseMap
Interface: Public

Answer the external object base map.

externalObjectBaseMap:

Date: Jul 26, 1990
Usage: externalObjectBaseMap: aDictionary
Interface: Public

Set the external object base map.

initialize

Date: Jul 26, 1990
Usage: initialize
Interface: Public

Set up initial instance variables.

map:to:

Date: Jul 26, 1990
Usage: map: anMDASElement to: objectBaseKey
Interface: Public

Map the MDAS-recognized element name to the actual element name.

mapFor:

Date: Jul 26, 1990
Usage: mapFor: anElement
Interface: Public

Answer the external object base map for an element.

objectBase

Date: Mar 21, 1990
Usage: objectBase
Interface: Public

Answer the object base.

objectBase:

Date: Mar 21, 1990
Usage: objectBase: anObjectBase
Interface: Public

Set the object base.

typeFor:

Date: Jul 26, 1990
Usage: typeFor: anElement
Interface: Public

Answer the external object base map for an element.

Class: MDASpreviousAction

Description:

This class holds on to a task that has been performed.

Instance Variables:

diagnostic - A snapshot of the diagnostic group before the task is performed.
task - A pointer to the task object (test or rect).
outcomes - Stores the observed outcomes for tests.
time - Stores the time to complete the task.

Instance Methods:

diagnostic

Date: Feb 6, 1990
Usage: diagnostic
Interface: Public

Answer the diagnostic state.

diagnostic:

Date: Feb 6, 1990
Usage: diagnostic: aDiagnostic
Interface: Public

Set the diagnostic state.

name

Date: Feb 6, 1990
Usage: name
Interface: Public

Answer the name of the task.

outcomes

Date: Feb 6, 1990
Usage: outcomes
Interface: Public

Answer the observed outcomes for the test tasks.

outcomes:

Date: Feb 6, 1990
Usage: outcomes: observedOutcomes
Interface: Public

Store the observed outcomes for tests.

task

Date: Feb 6, 1990
Usage: task
Interface: Public

Answer the task.

task:

Date: Feb 6, 1990
Usage: task: aTask
Interface: Public

Set the task to a rect or test.

time

Date: Feb 6, 1990
Usage: time
Interface: Public

Answer the time it took to perform the task.

time:

Date: Feb 6, 1990
Usage: time: aTime
Interface: Public

Set the time to the time it took to perform the task.

Class: MDASrankedTask

Description:

This class will store ranking information for the tasks.

Instance Variables:

task - A pointer to the task object (test or rect) to be ranked.
rankingValue - The value that ranking is based on.
taskProbability - The probability the task spans failed faults.
currenRanking - A number representing the task's current ranking.
critical - True or False depending if the task is critical.
type - T (test) or R (rect) for the type of task it is.

Class Methods:

new

Date: Feb 6, 1990
Usage: new
Interface: Public

Create a new instance of MDASrankedtask.

newWith:withType:

Date: Feb 6, 1990
Usage: newWith: aTask withType: aTaskType
Interface: Public

Create a new instance of MDASrankedtask.

Instance Methods:

available

Date: Feb 6, 1990
Usage: available
Interface: Public

Is task available? Return true until full presentation system is integrated.

critical

Date: Feb 6, 1990
Usage: critical
Interface: Public

Answer the criticality status of the task.

critical:

Date: Feb 6, 1990
Usage: critical: aBoolean
Interface: Public

Set the criticality status of the task.

currentRanking

Date: Feb 6, 1990
Usage: currentRanking
Interface: Public

Answer the current ranking status of the task.

currentRanking:

Date: Feb 6, 1990
Usage: currentRanking: aRanking
Interface: Public

Set the current ranking value of the task in interleaved.

degradedIsBetterThanRect:

Date: Mar 20, 1990
Usage: degradedIsBetterThanRect: aRankedRect
Interface: Public

Smart TOC rect ranking.

degradedIsBetterThanTest:

Date: Mar 20, 1990
Usage: degradedIsBetterThanTest: aRankedTest
Interface: Public

Smart TOC test ranking.

initializeWithType:

Date: Feb 6, 1990
Usage: initializeWithType: aTaskType
Interface: Public

Initialize the instance variables and set the task type, 'T' or 'R', for test or rect.

isBetterThanRect:

Date: Feb 6, 1990
Usage: isBetterThanRect: aRankedRect
Interface: Public

Compare self to a ranked rect. Answer true if better.

isBetterThanTask:

Date: Feb 6, 1990
Usage: isBetterThanTask: aRankedTask

Interface: Public

Compare self to a ranked task. Answer true if better.

isBetterThanTest:

Date: Feb 6, 1990
Usage: isBetterThanTest: aRankedTest
Interface: Public

Compare self to a ranked test. Answer true if better.

isRect

Date: Mar 6, 1990
Usage: isRect
Interface: Public

Answer True if task is a rect.

isTest

Date: Mar 6, 1990
Usage: isTest
Interface: Public

Answer True if task is a test.

name

Date: Feb 6, 1990
Usage: name
Interface: Public

Answer the name of the task.

printOn:

Date: Feb 6, 1990
Usage: printOn: aStream
Interface: Public

Modified the printOn: method for viewing.

probabilityOfFailure

Date: Feb 6, 1990
Usage: probabilityOfFailure
Interface: Public

Answer the probability of failure for a task.

probabilityOfFailure:

Date: Feb 6, 1990
Usage: probabilityOfFailure: aProbability
Interface: Public

Set the probability of failure for a task.

rankingValue

Date: Feb 6, 1990
Usage: rankingValue
Interface: Public

Answer the ranking value for a task.

rankingValue:

Date: Feb 6, 1990
Usage: rankingValue: aRankingValue
Interface: Public

Set the ranking value for a task.

repairLevel

Date: Feb 6, 1990
Usage: repairLevel
Interface: Public

Answer task repair level. Return 1 until presentation system is fully integrated.

task

Date: Feb 6, 1990
Usage: task
Interface: Public

Answer the task (Rectification or a Test).

task:

Date: Feb 6, 1990
Usage: task: aTask
Interface: Public

Set the task (Rectification or a Test).

time

Date: Feb 6, 1990
Usage: time
Interface: Public

Answer the time of the task (Rectification or a Test).

type

Date: Mar 6, 1990
Usage: type
Interface: Public

Answer the task type ('T' = test, 'R' = rect).

type:

Date: Mar 6, 1990
Usage: type: aTaskType
Interface: Public

Set the task type ('T' = test, 'R' = rect).

Class: MDASsets

Description:

This class keeps the state of the faults.

Instance Variables:

exculpatedFaults - Faults that are exculpated.
maybeSetFaults - Faults that are in the maybe set.
plausibleFaults - Faults that are plausible.
rectifiedFaults - Faults that are rectified.
isolatedFaults - Faults that are isolated.
unionFaults - Faults that are in the union.
criticalFaults - Faults that are critical.

Class Methods:

new

Date: Feb 6, 1990
Usage: new
Interface: Public

Create a new instance of MDASsets.

Instance Methods:

addToExculpated:

Date: Feb 7, 1990
Usage: addToExculpated: aFault

Interface: Public

Add a fault to the exculpated set. Also, if the fault was previously isolated, remove it from the isolated set as well.

addToIsolated:

Date: Feb 7, 1990
Usage: addToIsolated: aFault
Interface: Public

Add a fault to the isolated set.

addToMaybeSet:

Date: Feb 6, 1990
Usage: addToMaybeSet: aFault
Interface: Public

Add a fault to the maybeSet set.

addToPlausible:

Date: Feb 6, 1990
Usage: addToPlausible: aFault
Interface: Public

Add a fault to the plausible set.

addToRectified:

Date: Feb 6, 1990
Usage: addToRectified: aFault
Interface: Public

Add a fault to the rectified set. Also, if the fault was previously isolated, remove it from the isolated set as well.

addToUnion:

Date: Feb 6, 1990
Usage: addToUnion: aFault
Interface: Private

Add a fault to the union set.

anyCriticalsSet

Date: Feb 13, 1990
Usage: anyCriticalsSet
Interface: Public

Answer true if there are any criticals set; otherwise, answer false.

buildUnionSet:

Date: Feb 6, 1990
Usage: buildUnionSet: observedSymptoms
Interface: Public

Build the union set from the list of observed symptoms.

checkIfCritical:

Date: Feb 6, 1990
Usage: checkIfCritical: aFaultList
Interface: Public

Set the criticality status of the fault list.

checkIfCriticalRect:

Date: May 22, 1990
Usage: checkIfCriticalRect: aFaultList
Interface: Public

Set the criticality status for the spanned faults of a rectification.

checkIfCriticalTest:

Date: May 22, 1990
Usage: checkIfCriticalTest: aFaultList
Interface: Public

Set the criticality status for the spanned faults of a test.

clearAllMaybeSet

Date: Feb 6, 1990
Usage: clearAllMaybeSet
Interface: Public

Clear all maybeSet faults.

clearAllPlausible

Date: Feb 6, 1990
Usage: clearAllPlausible
Interface: Public

Clear all plausible faults.

clearAllSetsOf:

Date: Feb 6, 1990
Usage: clearAllSetsOf: aFault
Interface: Private

Clear all sets except isolated, union, and critical faults of a fault.

clearAllUnion

Date: Feb 6, 1990
Usage: clearAllUnion
Interface: Private

Clear all union faults.

clearIsolatedOf:

Date: Feb 7, 1990
Usage: clearIsolatedOf: aFault
Interface: Private

Clear isolated faults of a fault.

criticalFaults

Date: Feb 6, 1990
Usage: criticalFaults
Interface: Public

Answer the set of critical faults based on the union.

deepcopy

Date: Feb 6, 1990
Usage: deepcopy
Interface: Public

Answer a copy of the receiver with deep copies of each instance variable.

exculpatedFaults

Date: Feb 6, 1990
Usage: exculpatedFaults
Interface: Public

Answer the set of exculpated faults.

initialize

Date: Feb 6, 1990
Usage: initialize
Interface: Private

Initialize the needed instance variables.

isolatedFaults

Date: Feb 7, 1990
Usage: isolatedFaults

Interface: Public

Answer the isolated faults list.

maybeSetFaults

Date: Feb 6, 1990
Usage: maybeSetFaults
Interface: Public

Answer the set of maybeSet faults.

numberOfCriticalFaults

Date: Feb 6, 1990
Usage: numberOfCriticalFaults
Interface: Public

Answer the number of critical faults not exculpated.

numberOfMaybeSetFaults

Date: Feb 6, 1990
Usage: numberOfMaybeSetFaults
Interface: Public

Answer the number of maybeSet faults.

numberOfPlausibleFaults

Date: Feb 6, 1990
Usage: numberOfPlausibleFaults
Interface: Public

Answer the number of plausible faults.

plausibleFaults

Date: Feb 6, 1990
Usage: plausibleFaults
Interface: Public

Answer the set of plausible faults.

rectifiedFaults

Date: Feb 6, 1990
Usage: rectifiedFaults
Interface: Public

Answer the set of rectified faults.

setCriticalFaults:

Date: Feb 6, 1990
Usage: setCriticalFaults: criticalFaultsList
Interface: Public

Set the critical faults that are plausible and answer the number of critical faults.

unionFaults

Date: Feb 6, 1990
Usage: unionFaults
Interface: Public

Answer the set of union faults.

Class: MDAStasks

Description:

This class keeps the tasks (tests or rects) for a diagnostic group.

Instance Variables:

rects - A list of rectifications.
tests - A list of tests.
usefulRects - A list of the rects that are still useful.
usefulTests - A list of the tests that are still useful.
rankedRects - A list of ranked tasks for useful rectifications.
rankedTests - A list of ranked tasks for useful tests.
interleavedTasks - A list of ranked tasks for useful rectifications and tests.

Class Methods:

new

Date: Feb 6, 1990
Usage: new
Interface: Public

Create a new instance of MDAStasks.

Instance Methods:

addAllToRects:

Date: Feb 6, 1990
Usage: addAllToRects: aCollection
Interface: Public

Add all rectifications in a collection to the rects set.

addAllToTests:

Date: Feb 6, 1990
Usage: addAllToTests: aCollection
Interface: Public

Add all tests in a collection to the tests set.

addToInterleavedTasks:

Date: Feb 6, 1990
Usage: addToInterleavedTasks: aRankedTask
Interface: Public

Add a ranked task to the interleaved tasks list.

addToRankedRects:

Date: Feb 6, 1990
Usage: addToRankedRects: aRankedRect
Interface: Public

Add a ranked rectification to the ranked rects list.

addToRankedTests:

Date: Feb 6, 1990
Usage: addToRankedTests: aRankedTests
Interface: Public

Add a ranked test to the ranked tests list.

addToRects:

Date: Feb 6, 1990
Usage: addToRects: aRect
Interface: Private

Add a rectification to the rects set.

addToTests:

Date: Jan 17, 1990
Usage: addToTests: aTest
Interface: Private

Add a test to the tests set.

deepcopy

Date: Feb 6, 1990
Usage: deepcopy
Interface: Public

Answer a copy of the receiver with deep copies of each instance variable.

initialize

Date: Feb 6, 1990
Usage: initialize
Interface: Private

Initialize the needed instance variables.

interleavedTasks

Date: Feb 6, 1990
Usage: interleavedTasks
Interface: Public

Answer the interleaved tasks list.

rankedRects

Date: Feb 6, 1990
Usage: rankedRects
Interface: Public

Answer the ranked rects list.

rankedTests

Date: Feb 6, 1990
Usage: rankedTests
Interface: Public

Answer the ranked tests list.

rects

Date: Feb 6, 1990
Usage: rects
Interface: Public

Answer the list of rects.

setRankedLists

Date: Feb 6, 1990
Usage: setRankedLists
Interface: Public

Initialize all ranked lists with the appropriate sort blocks.

setUsefulRects

Date: Feb 6, 1990

Usage: setUsefulRects
Interface: Public

Make all applicable rectifications useful.

setUsefulTests

Date: Feb 6, 1990
Usage: setUsefulTests
Interface: Public

Make all applicable tests useful.

tests

Date: Feb 6, 1990
Usage: tests
Interface: Public

Answer the list of tests.

usefulRects

Date: Feb 6, 1990
Usage: usefulRects
Interface: Public

Answer the useful rects list.

usefulTests

Date: Feb 6, 1990
Usage: usefulTests
Interface: Public

Answer the useful tests list.

Class: MDASphysicalController

Class Methods:

new

Date: Jul 24, 1990
Usage: new
Interface: Public

Create a new instance of MDASphysicalController.

Instance Methods:

addFunctionalCheck:

Date: Aug 1, 1990
Usage: addFunctionalCheck: aTest
Interface: Private

Add a test to the list of functional tests to perform.

addHazards:at:

Date: Sept 5, 1990
Usage: addHazards: aHazardCollection at: aLocation
Interface: Public

Add hazards found in a location to a location key within a dictionary.

affectRectList

Date: Aug 10, 1990
Usage: affectRectList
Interface: Public

Load the affect rects list dictionary.

affectRepair:

Date: Sept 5, 1990
Usage: affectRepair: aRect
Interface: Public

Start affect component repair.

affectRepairOther:

Date: Sept 5, 1990
Usage: affectRepairOther: aRect
Interface: Public

Isolate and repair affected components that were suspected or damaged.

answerMessage

Date: Sept 4, 1990
Usage: answerMessage
Interface: Public

Returns menu-selected information to the next physical model method.

answerMessage:

Date: Sept 4, 1990
Usage: answerMessage: aMessageSelector

Interface: Public

Set menu-selected information to return to the next physical model method.

bestTask

Date: Jul 24, 1990
Usage: bestTask
Interface: Public

Answer the no task from smart TOC in physical mode.

cleanUpAndEvaluateMA

Date: Aug 13, 1990
Usage: cleanUpAndEvaluateMA
Interface: Public

Display a message to clean up with a maintenance action and repair when required.

clearPromptMessages

Date: Jul 31, 1990
Usage: clearPromptMessages
Interface: Public

Clear the message-passing prompts.

currentHazardChooser

Date: Sept 5, 1990
Usage: currentHazardChooser
Interface: Public

Prompt the user for current hazard to investigate.

currentLocation

Date: Jul 31, 1990
Usage: currentLocation
Interface: Public

Answer the current location.

currentLocation:

Date: Jul 31, 1990
Usage: currentLocation: aLocation
Interface: Public

Set the current location.

currentRect

Date: Sept 4, 1990
Usage: currentRect
Interface: Public

Return the current rectification from the physical module.

damageAssessmentChooser

Date: Sept 4, 1990
Usage: damageAssessmentChooser
Interface: Public

Display rectifications and damages codes for selection by the technician in the event that there are source components to investigate.

damagedOrSuspectedComponentChooser

Date: Sept 4, 1990
Usage: damagedOrSuspectedComponentChooser
Interface: Public

Display rectifications and damages codes for selection by the technician in the event that there are damaged or or supected affected components.

destroyedComponentChooser

Date: Sept 4, 1990
Usage: destroyedComponentChooser
Interface: Public

Display destroyed component rectifications for selection and repair by the technician.

diagnosticModeStatus

Date: Sept 4, 1990
Usage: diagnosticModeStatus
Interface: Public

Return the current diagnostic physical status (source/effects) from the physical module.

Doit

No documentation available.

dynamicDistroyedCompChooser

Date: Sept 5, 1990
Usage: dynamicDistroyedCompChooser
Interface: Public

Display source rectifications for dynamic inspection and repair by the technician.

dynamicSourceRepair:

Date: Sept 5, 1990
Usage: dynamicSourceRepair: aRect
Interface: Public

Prompt physical module to continue dynamic source repair and to delete rectifications that have been performed.

effectDegreeOfDamage:

Date: Sept 5, 1990
Usage: effectDegreeOfDamage: aDamageCode
Interface: Public

Assess degree of damage to affected components.

extractAppropriateRectsFrom:

Date: Sept 5, 1990
Usage: extractAppropriateRectsFrom: aRectsCollection
Interface: Public

Set the spanned physical hazard elements for the physical module.

functionalCheck

Date: Sept 5, 1990
Usage: functionalCheck
Interface: Public

Display a functional check and prompt the technician for results.

functionalChecks

Date: Aug 1, 1990
Usage: functionalChecks
Interface: Public

Answer fuctional check collection.

functionalChecks:

Date: Aug 1, 1990
Usage: functionalChecks: anOrderedCollection
Interface: Public

Set an empty functional check collection.

functionalEntry

Date: Sept 5, 1990
Usage: functionalEntry
Interface: Public

Begin physical diagnostics from functional diagnostics.

functionalRects

Date: Aug 6, 1990
Usage: functionalRects
Interface: Public

Answer the current functional rectification.

functionalRects:

Date: Aug 6, 1990
Usage: functionalRects: aRectList
Interface: Public

Set the current functional list.

functionalTest:

Date: Sept 5, 1990
Usage: functionalTest: aTestResult
Interface: Public

Perform a functional test and return the results to the physical module.

initialize

Date: Jul 31, 1990
Usage: initialize
Interface: Public

Initialize the physical controller and physical model logic.

initializeFromPhysicalStart:

Date: Sept 5, 1990
Usage: initializeFromPhysicalStart: aResult
Interface: Public

Start the physical model after functional model system returns a pass on the functional check.

initializePhysicalModelLogic

Date: Sept 5, 1990
Usage: initializePhysicalModelLogic
Interface: Public

Initialize the physical model's logic data base.

initializePhysicalModule

Date: Sept 5, 1990
Usage: initializePhysicalModule
Interface: Public

Initialize the physical module's data attributes and start the physical module.

isolateAndRepairAffects:

Date: Sept 5, 1990
Usage: isolateAndRepairAffects: x
Interface: Public

Isolate and repair or continue isolation and repair of affected components.

isolateAndRepairSources:

Date: Sept 5, 1990
Usage: isolateAndRepairSources: passFail
Interface: Public

Isolate and repair sources based on the functional check outcome (pass/fail).

isRepairRequired

Date: Sept 5, 1990
Usage: isRepairRequired
Interface: Public

Prompt the technician for a response to (Is repair required?).

isRepairRequiredResponse:

Date: Sept 5, 1990
Usage: isRepairRequiredResponse: yesNo
Interface: Public

Respond with the next method based on the response to the question whether repair is required.

jumpToFunctionalMode

Date: Aug 1, 1990
Usage: jumpToFunctionalMode
Interface: Public

Inform diagnostics to pass control to the functional model.

limitedOPSCheck

Date: Jul 31, 1990
Usage: limitedOPSCheck
Interface: Public

Prompt technician for results of a limited OPS check.

limitedOPSCheckDesireability:

Date: Jul 31, 1990
Usage: limitedOPSCheckDesireability: yesNo
Interface: Public

If a limited ops check is desired, get the results. Otherwise, use pass.

limitedOPSCheckDesired

Date: Jul 31, 1990
Usage: limitedOPSCheckDesired
Interface: Public

Prompt technician for desirability of a limited OPS check.

locations

Date: Jul 31, 1990
Usage: locations
Interface: Public

Answer a location collection.

locations:

Date: Jul 31, 1990
Usage: locations: aCollection
Interface: Public

Set an empty location collection.

menuLabelArray

Date: Sept 5, 1990
Usage: menuLabelArray
Interface: Public

Answer the menu label for the next menu.

menuLabelArray:

Date: Sept 5, 1990
Usage: menuLabelArray: aLabelArray
Interface: Public

Set an empty menu label array.

menuLabelWith

Date: Sept 5, 1990
Usage: menuLabelWith
Interface: Public

Answer a menu of label array.

menuLabelWith:

Date: Sept 5, 1990
Usage: menuLabelWith: aLabelSelector
Interface: Public

Set an empty menu with label array.

menuSelectors

Date: Sept 5, 1990
Usage: menuSelectors
Interface: Public

Answer the menu item selectors.

menuSelectors:

Date: Sept 5, 1990
Usage: menuSelectors: aSelectorSet
Interface: Public

Set the empty menu item selectors.

menuTitle

Date: Sept 5, 1990
Usage: menuTitle
Interface: Public

Answer the menu title.

menuTitle:

Date: Sept 5, 1990
Usage: menuTitle: aString
Interface: Public

Set the empty menu title.

modifyEffectProbabilities:

Date: Sept 5, 1990
Usage: modifyEffectProbabilities: passFail

Interface: Public

Respond with the next method, based on the outcome of the functional check or limited OPS check. A pass will continue physical assessment; a fail will modify probabilities.

modifySourceProbabilities:

Date: Sept 5, 1990
Usage: modifySourceProbabilities: passFail
Interface: Public

If a fail is exhibited on the functional check, rectifications with damage codes are passed to the functional module; if a pass is exhibited, physical assessment continues.

mustSelect

Date: Sept 5, 1990
Usage: mustSelect
Interface: Public

Answer a Boolean message for must select menus.

mustSelect:

Date: Sept 5, 1990
Usage: mustSelect: aBoolean
Interface: Public

Set an empty Boolean variable for must select menus.

next

Date: Jul 27, 1990
Usage: next
Interface: Public

Answer what the next menu and method are.

nextMethod

Date: July 31, 1990
Usage: nextMethod
Interface: Public

Answer the next logic method.

nextMethod:

Date: July 31, 1990
Usage: nextMethod: aSymbol
Interface: Public

Initialize the next logic method.

nextTask

Date: Jul 31, 1990
Usage: nextTask
Interface: Public

Answer the nextTask.

nextTask:

Date: Jul 31, 1990
Usage: nextTask: aTask
Interface: Public

Set the nextTask.

objectBase

Date: Mar 21, 1990
Usage: objectBase
Interface: Public

Answer the object base.

objectBase:

Date: Sept 5, 1990
Usage: objectBase: anObjectBase
Interface: Public

Set the object base.

otherEffectRepair:

Date: Sept 5, 1990
Usage: otherEffectRepair: aRect
Interface: Public

Isolate and repair effects that are suspected or damaged.

performRectification:

Date: Sept 5, 1990
Usage: performRectification: aRect
Interface: Public

Display a rectification.

physHazardGroups

Date: Apr 6, 1990
Usage: physHazardGroups

Interface: Public

Answer the physical hazard groups sets.

physHazardGroups:

Date: Apr 6, 1990
Usage: physHazardGroups: aDictionary
Interface: Public

Set the physical hazard groups sets.

physicalModelLogic

Date: Jul 24, 1990
Usage: physicalModelLogic
Interface: Public

Answer the physical model logic.

physicalModelLogic:

Date: Jul 24, 1990
Usage: physicalModelLogic: aPhysicalModelLogic
Interface: Public

Set the physical model logic.

presentationMessage

Date: Sept 5, 1990
Usage: presentationMessage
Interface: Public

Answer the next presentation message.

presentationMessage:

Date: Sept 5, 1990
Usage: presentationMessage: aMessageSelector
Interface: Public

Set the next presentation message.

presentMenu

Date: July 17, 1990
Usage: presentMenu
Interface: Public

Answer the presentation menu.

presentMenu:

Date: Sept 5, 1990
Usage: presentMenu: aSymbol
Interface: Public

Initialize the next presentation menu.

quit

Date: Aug 3, 1990
Usage: quit
Interface: Public

The physical model is complete.

rectProbabilityModifiers

Date: Sept 5, 1990
Usage: rectProbabilityModifiers
Interface: Public

Return physical rects with damage codes to the functional module.

removeEffectRect:

Date: Sept 5, 1990
Usage: removeEffectRect: aRect
Interface: Public

Remove the effect rectification performed from the physical logic dictionary.

removeSourceRect:

Date: Sept 5, 1990
Usage: removeSourceRect: aRect
Interface: Public

Remove the source rectification performed from the physical logic dictionary.

selectCurrentHazard:

Date: Sept 5, 1990
Usage: selectCurrentHazard: x
Interface: Public

Select the next current hazard to investigate.

selectedHazards:

Date: Jul 31, 1990
Usage: selectedHazards: aCollection
Interface: Public

Set a collection of selected hazards for the current location.

selectedLocations:

Date: Sept 5, 1990
Usage: selectedLocations: aCollection
Interface: Public

Set a collection of locations and select hazards for each location.

selectHazardsForCurrentLocation

Date: Sept 5, 1990
Usage: selectHazardsForCurrentLocation
Interface: Public

Prompt the user for all the hazards observed at the current location.

selectLocations

Date: Sept 5, 1990
Usage: selectLocations
Interface: Public

Prompt the user for all the locations in which hazards are observed.

selectOne

Date: Sept 5, 1990
Usage: selectOne
Interface: Public

Answer a Boolean message for menus requiring only one selection response.

selectOne:

Date: Sept 5, 1990
Usage: selectOne: aBoolean
Interface: Public

Set the empty Boolean message for menus requiring only one selection response.

sourceControllerHazards

Date: Sept 5, 1990
Usage: sourceControllerHazards
Interface: Public

Load the hazard list dictionary.

sourceDegreeOfDamage:

Date: Sept 5, 1990

Usage: sourceDegreeOfDamage: aDamageCode
Interface: Public

Continue selecting component source degree of damage.

sourceRectList

Date: Aug 10, 1990
Usage: sourceRectList
Interface: Public

Load the source rects list dictionary.

sourceRepair:

Date: Sept 5, 1990
Usage: sourceRepair: aRect
Interface: Public

Continue source repair of components.

sourceSystemPerformanceChooser

Date: Aug 6, 1990
Usage: sourceSystemPerformianceChooser
Interface: Public

Display a message to determine if the source system is needed for functional OPS check.

sourceSystemResult:

Date: Sept 5, 1990
Usage: sourceSystemResult: yesNo
Interface: Public

Begin physical assessment from functional diagnostics.

startPhysicalAssessment:

Date: Sept 5, 1990
Usage: startPhysicalAssessment: aFunctionalRectsList
Interface: Public

Start the physical assessment model by initializing the location and hazard attributes.

taskMessage

Date: Sept 5, 1990
Usage: taskMessage
Interface: Public

Answer a task message.

taskMessage:

Date: Sept 5, 1990
Usage: taskMessage: aMessageSelector
Interface: Public

Set the next task message.

visuallyDistroyedCompChooser

Date: Aug 13, 1990
Usage: visuallyDistroyedCompChooser
Interface: Public

Display rectifications for visual inspection and repair by the technician.

whatsNext

Date: Jul 26, 1990
Usage: whatsNext
Interface: Private

Answer the description of the thing to do next.

Class: PhysModel

Instance Methods:

affectDegreeOfDamage:

Date: Aug 9, 1990
Usage:
Interface: Public

Determine degree of affected component damage.

affectRectsPerformed:

Date: Sept 5, 1990
Usage:
Interface: Private

Delete affected rectifications performed.

affectRepair:

Date: Sept 5, 1990
Usage:
Interface: Public

Repair affected component damage by damage code.

anIntersection:

Date: Aug 9, 1990
Usage:
Interface: Private

Answer intersecting members of two Lists.

areAffectRectsRepaired:

Date: Sept 5, 1990
Usage:
Interface: Private

Assign repaired affected components from dictionary a damage code value of 1.00 (no apparent effect).

areSourceRectsRepaired:

Date: Sept 5, 1990
Usage:
Interface: Private

Put repaired source components into a rectified set.

clearCompsAffected:

Date: Sept 5, 1990
Usage:
Interface: Private

Delete affected components which were confirmed good from a functional check pass.

clearCurrentAffects:

Date: Aug 9, 1990
Usage:
Interface: Private

Delete affected repaired components which were confirmed good from a functional check pass.

clearCurrentSources:

Date: Aug 9, 1990
Usage:
Interface: Private

Delete source components which were confirmed good from a functional check pass.

clearModProbData:

Date: Sept 5, 1990
Usage:
Interface: Private

Reset rectified component dictionaries for modify probabilities.

currentAffHazards:

Date: Aug 9, 1990
Usage:
Interface: Private

Answer affected components which are within the current system.

currentSrceHazards:

Date: Sept 5, 1990
Usage:
Interface: Private

Answer source components which are within the current system.

destroyedAffectRepair:

Date: Aug 9, 1990
Usage:
Interface: Private

Repair destroyed affected components.

dynamicSourceRepair:

Date: Sept 5, 1990
Usage:
Interface: Public

Repair source components during dynamic inspection.

effects:

Date: Aug 9, 1990
Usage:
Interface: Private

The rectHaz includes the observed accessGrp and vulnerable hazards.

effectsRemaining:

Date: Aug 9, 1990
Usage:
Interface: Private

Check to see if any affected systems remain that have not been rectified.

fromFunctFC:

Date: Aug 9, 1990
Usage:
Interface: Public

Determine if affected components are in current system and proceed with physical or functional diagnostics.

functSysOnEffectsList:

Date: Sept 5, 1990
Usage:
Interface: Private

Determine if affected components are in current system from functional module.

functSysOnSourceList:

Date: Aug 9, 1990
Usage:
Interface: Private

Determine if source and/or affected components are in current system and proceed with physical or functional diagnostics.

initCurrentHazardGroup:

Date: Aug 9, 1990
Usage:
Interface: Private

Initialize current source and effects data for the hazard chosen.

initialize:

Date: Sept 5, 1990
Usage:
Interface: Public

Initialize hazard effects and sources sets.

initializeEffects:

Date: Aug 9, 1990
Usage:
Interface: Private

Initialize hazard effects sets.

initializeFromPhysicalStart:

Date: Aug 9, 1990
Usage:
Interface: Public

Initialize sources, effects, and affected systems.

initializeHazards:

Date: Aug 9, 1990
Usage:
Interface: Public

Initialize controller hazard sets for functional interface.

initializeNewEffect:

Date: Sept 5, 1990
Usage:
Interface: Public

Initialize effect controller rectification sets for functional interface.

initializeNewSource:

Date: Aug 9, 1990
Usage:
Interface: Public

Initialize source controller rectification sets for functional interface.

initializeSources:

Date: Sept 5, 1990
Usage:
Interface: Private

Initialize hazard sources sets.

isolateAndRepairEffects:

Date: Aug 9, 1990
Usage:
Interface: Public

Isolate and repair effects of hazards by system.

isolateAndRepairSources:

Date: Sept 5, 1990
Usage:
Interface: Public

Isolate and repair sources of hazards by hazard selected and system.

loadSmalltalkEffectData:

Date: Aug 9, 1990
Usage:
Interface: Private

Load data for effect repair chooser.

loadSmalltalkSourceData:

Date: Aug 9, 1990
Usage:
Interface: Private

Load data for source repair chooser.

member:

Date: May 10, 1990
Usage:
Interface: Private

Succeed if the first argument is a member of a list in the second argument.

modifyEffectProbabilities:

Date: Sept 5, 1990
Usage:
Interface: Public

Load affected components with damage codes and prompt the functional assessment module to continue diagnostics.

modifySourceProbabilities:

Date: Sept 5, 1990
Usage:
Interface: Public

Load source components and damage codes and prompt the functional assessment module to continue diagnostics.

moreEffects:

Date: Aug 9, 1990
Usage:
Interface: Private

Prompt the data to see if the data base contains affected systems that have not been repaired. Also reinitialize data elements and repeat.

moreHazardSources:

Date: Sept 5, 1990
Usage:
Interface: Private

Prompt the data to see if the data base contains source hazards that have not been repaired. Also reinitialize data elements and repeat.

moreSources:

Date: Sept 5, 1990
Usage:
Interface: Private

Prompt the data to see if the data base contains source systems and hazards that have not been repaired. Also reinitialize data elements and repeat.

moreSystemSources:

Date: Aug 9, 1990
Usage:
Interface: Private

Prompt the data to see if the data base contains source systems that have not been repaired. Also reinitialize data elements and repeat.

noMoreEffects:

Date: Aug 9, 1990
Usage:
Interface: Private

If there are no more affected systems to investigate, physical assessment is ended.

noMoreSources:

Date: Aug 9, 1990
Usage:
Interface: Private

Continue to isolate and repair effects if there are not more source systems present.

otherAffectRepair:

Date: Sept 5, 1990
Usage:
Interface: Public

Repair other affected component damage.

otherAffectRepairChooser:

Date: Sept 5, 1990
Usage:
Interface: Private

Repair affected components that are either damaged or suspected.

repairsMade:

Date: Aug 9, 1990
Usage:
Interface: Private

If repairs have been made, the user is prompted for the performance and outcome of a functional check.

repairsNotMade:

Date: Sept 5, 1990
Usage:
Interface: Private

Check to see if repairs have been made and if not, prompt the user to see if a limited OPS check is desired.

selectCurrentHazard:

Date: Sept 5, 1990
Usage:
Interface: Public

Initialize current hazard information and store other hazard information for future evaluation.

selectSourceHazards:

Date: Aug 9, 1990
Usage:
Interface: Private

Isolate and repair sources of hazards by system.

sourceDegreeOfDamage:

Date: Sept 5, 1990
Usage:
Interface: Public

Determine degree of source damage.

sourceRectsPerformed:

Date: Sept 5, 1990

Usage:
Interface: Private

Repair source damage.

sourceRepair:

Date: Sept 5, 1990
Usage:
Interface: Public

Inspect sources of hazards by system.

sources:

Date: Sept 5, 1990
Usage:
Interface: Private

The rectHaz includes the observed location and contained hazard.

visualSourceRepair:

Date: Sept 5, 1990
Usage:
Interface: Private

Repair source components with visual damage.

* U. S. GOVERNMENT PRINTING OFFICE: 1990--561-052/20179