

GCESS: A Symptom Driven Diagnostic Shell and Related Applications

Peter Holtzman and Ray Fischer

Inference Corporation
550 North Continental Blvd
El Segundo, CA 90245
e-mail: holtzman@inference.com

Northrop Corporation
2301 West 120th St.
Hawthorne, CA 90251

Abstract

An application shell is presented for symptom driven diagnostic applications that is designed to support multiple diagnostic paradigms and is used for several specific diagnostic applications.

1. Introduction

The United States Air Force deploys Peacekeeper missiles as part of its strategic forces. Guidance and control components are routinely tested at missile sites. If they fail these field tests they are sent to an Air Force repair facility (the Aerospace Guidance and Metrology Center). The components are electronic and electro mechanical in nature, and are manufactured by a number of different defense contractors. The repair facility must diagnose and then repair the problem.

Typically, faulty components arrive at the repair facility with little information about the faulty behavior. They are subjected to extended acceptance tests to verify that problems exist, and in some cases, to special purpose testing. Acceptance tests fail when parameters fall outside specified limits. Testing is performed on automatic test equipment (ATE) and may take many hours and, occasionally, days. It generates very large amounts of (sensor) data that must be examined to diagnose the failure.

In many cases, the Air Force personnel must rely on the contractors' experts to isolate the fault. This involves additional expense and delay. The Air Force would like to perform as much of the diagnosis itself as possible, and acquire the contractors' expertise.

Currently, much of the expert knowledge of the components is disappearing. This is to be expected in a long lived program, as the original designers and engineers retire or leave. It has been accelerated as political changes have impacted personnel levels.

Some diagnostic programs exist on the ATEs that report anomalies and attempt to identify faults and

repair actions. Problems associated with such software, as explained below, have resulted in its general disuse, causing contractors to consider alternative means of replacing these programs. Since AI has produced many diagnostic systems, several contractors independently suggested building diagnostic expert systems. These suggestions led the Ballistic Missile Organization (BMO) to build the Guidance and Control Expert System Shell (GCESS).

Problems with Existing Software

As an example of existing software we examine the existing Inertial Measurement Unit (IMU) diagnostics, since the IMU was the first area we used to prototype a new diagnostic approach.

The IMU for the Peacekeeper was developed by Northrop Corporation. Along with the IMU, Northrop was required to develop diagnostic software to analyze the acceptance test data. The software was designed along with the unit. While the purpose of the software was to assure ease of maintenance, it required Northrop to predict with certainty the behavior of a complex physical component, before acquiring actual experience using it. These predictions, although sound, were not absolutely correct and could not have been expected to be so. It is necessary to actually build and test a component in order to accurately understand its actual behavior. To build correct diagnostics, you must not only know the actual behavior of a correctly functioning unit, but also the actual behavior of a faulty unit. This requires real experience with functioning units.

From a user perspective, the major failings of the diagnostic program were that it could take several hours to run, frequently indicated incorrect repair recommendations, and did not produce information that would permit technicians to evaluate the reasonableness of the recommendation. The program was also difficult and costly to maintain, and could only be updated infrequently.

The Northrop approach was to predict all failure signatures and then construct a program to search for all such signatures in the data, independent of the acceptance test failures (symptoms). The program halted after the first signature it found. As it turned out, some of the signatures could be triggered without affecting the basic performance, and without relationship to any true symptom. This could not have been predicted theoretically.

Northrop engineers, in effect, used an informal causal mental model to predict the observable result of a given fault (a signature). They traced the causal network backward from effect to cause, and associated the faults as possible causes of a signature. This information was used as the specification for the diagnostic program that was produced by the programming staff. In disregarding the symptoms, the program behaved like a doctor performing a checkup. In principle, it should have always uncovered a fault, but in practice it did not.

In the case of the IMU diagnostics, the program specifications developed by the engineering staff comprised several hundred pages. Many of the signatures used variables that represented intermediate calculations. It was clear that it would not be an easy task for an engineer to readily change the specification. Because of the use of intermediate calculations, it could be anticipated that even several minor specification changes could entail considerable maintenance programming costs, and involve extensive regression testing.

It might seem as though the maintenance problems could be greatly ameliorated by the use of conventional software, a symptom driven method, and modern software engineering practices. Upon further analysis, it became clear that they were, in large part, a result of the conventional software block change maintenance cycle.

The software maintenance problems are strongly related to the fact that the diagnostics execute on the ATE. The ATE is used for acceptance testing of nuclear missile parts, and its software is subjected to very rigorous testing and configuration control. Individual change requests are accumulated and performed as part of infrequent and costly block changes. This is dictated by the organizational process of changing ATE software, rather than the characteristics of the software or changes.

The Diagnostic Life Cycle and Decision to Build An AI Diagnostic Shell

Given the problems with existing software, several contractors independently proposed building their

own diagnostic program based upon AI technology. The Ballistic Missile Organization (BMO) decided instead to build a single symptom driven diagnostic shell to be used by all contractors. This shell would be tailored to construct diagnostic applications as distinguished from more general AI expert system shells. This was based partly on the desire to avoid duplication of effort and cost, and partly on the desire to deliver a single shell to the repair facility personnel with a consistent cognitive model and user interface, rather than disparate tools developed by each contractor. It also allowed BMO to redress the major failure of the conventional approach, and address the broader problem of establishing a repository of diagnostic knowledge that could be transferred from the contractors to the Air Force.

BMO was keenly aware that the major failing of the conventional approach was in the area of ease of development and maintenance of diagnostic procedures. The Peacekeeper experience illustrated that in the beginning of the IMU production program there was a rapid evolution of diagnostic procedures, as experience accumulated about the actual behavior and problems with components in the field. It took some time for the diagnostics to "mature."

After that, diagnostics typically undergo a gradual development. These could be attributed to diagnostic discoveries, the appearance of new problems as components aged, and change in diagnostics as sub components were replaced with slightly different configurations. The changes need not impact the acceptance testing, so that the basic functions of the ATE programs did not need to be updated.

Human technicians could easily accommodate the *diagnostic life cycle* of rapid change followed by much more gradual change, but the conventional software maintenance process could not implement changes in a timely manner. This led to major differences between the technicians practice and the diagnostic programs, which condemned to software to go unused.

AI technology was viewed as potentially capable of supplying an easily maintainable shell that would allow engineers and technicians to maintain the diagnostics without programming staff and the delays and costs of the traditional software maintenance cycle. It would also provide a practical repository of diagnostic knowledge.

BMO was interested in a shell to support its various components and component manufacturing contractors. It wanted the shell to be capable of incorporating multiple diagnostic paradigms that

might arise from the different contractors individual diagnostic approaches.

The key organizational or process feature that would allow the development of this type of shell was the separation of acceptance testing from diagnosis, by hosting the diagnosis on a separate computer. This relieved the diagnostic software from the intensive and rigorous configuration control associated with acceptance testing. The PC was selected as the diagnostic hardware platform as a low cost delivery vehicle.

A key technical feature would be to modularize diagnostic knowledge and make it directly available for the engineering staff to maintain.

2. The GCESS System Overview

Development History

The Air Force contracted Northrop Corporation to build a prototype Guidance and Control Expert System Shell (GCESS) and to prototype the IMU diagnostic application. Northrop could not find an off-the-shelf shell that satisfied the Air Force requirements, selected ART-IM to construct the GCESS shell, and subcontracted Inference Corporation to build it. The diagnostic paradigm was developed jointly by Northrop and Inference. The selection of ART-IM was motivated by its extensive frame and rule system, availability on the PC platform, and ability to produce both development and deployed versions of expert systems.

The prototype shell [3] required nine man months of effort to build. The prototype IMU application illustrated the feasibility of the approach, and the advantages of using a declarative representation. In the development mode, the updating of diagnostic knowledge was almost immediate, and the application could be immediately rerun.

The Air Force contracted Northrop to build the actual shell, and a full IMU diagnostic system. Northrop subcontracted to Inference to develop diagnostic paradigms that would satisfy the Peacekeeper guidance and control community, including associated contractors. The GCESS shell required three man years of effort and was delivered at the end of 1991. Part of the work on the shell was to establish what diagnostic paradigms were needed by the contractors. The Northrop prototype application was expanded to a full IMU diagnostic system by Northrop engineers, and first deployed in June 1992. GCESS was developed as a Microsoft Windows application, to allow it to be run as one of several possible windowed processes. This allowed

simultaneous access to signal processing and networked data base applications.

GCESS initially supported two distinct diagnostic paradigms. One was developed by Northrop and Inference, and a separate paradigm was developed by Rockwell International that was prototyping diagnostics for test stations themselves.

The Air Force funded additional work on GCESS and a second version of the GCESS was delivered in December 1992. This version included an enhanced graphics browser, enhanced facilities to locate anomalous fault classes, and several other features, and was upgraded to use Microsoft Windows 3.1 features.

The second version does not include the paradigm developed for test station diagnostics, at the direction of the Air Force. However, both paradigms have been tested and the multiple diagnostic paradigm capability validated.

The development of the GCESS involved production of a requirement, functional specifications, high level design, design, and test plan documents. These documents and their various traceability matrices ensured that the requirements were addressed and tested.

The IMU application was tested against historical data on actual component failures, and correctly diagnosed all faults associated with those failures.

Several other diagnostic applications have been developed using GCESS. They include a second IMU diagnostic application for field data, and the use of a GCESS application as a component in a fleet assessment tool developed by Rockwell International.

GCESS Design Goals

The major GCESS design goals were to:

- Incorporate Multiple Symptom Driven Paradigms.
- Allow Multiple Fault Diagnosis for Each Symptom.
- Enable Declarative Representation of Diagnostic Knowledge.
- Provide Rich Explanation Facilities.
- Simplify Ease of Maintenance and Incremental Modification.
- Support a Deployment Mode which Inhibits Modifications.
- Separate Acceptance Testing from Diagnosis.
- Separate Diagnostic Specification from Data Specification and Retrieval.

The unsuccessful experience with the existing (non-symptom driven) software resulted in the goals of establishing a symptom driven system, capable of isolating multiple faults per symptom, and an architecture capable of incorporating more than one diagnostic paradigm. The use of declarative knowledge and rich explanation facilities would permit the development of diagnostics and aid the ease of maintainability, by the engineers themselves and also play a major role in the acceptance of the software by the technicians. Equally importantly, it would provide a repository of diagnostic knowledge that could be easily transferred to the Air Force. Ease of modification was an important goal. At the same time, the Air Force wanted to provide a "deployment" version of the software to be used by the technicians on the floor and which could not be modified, in order to provide adequate version control. Separation of acceptance testing from diagnosis has already been mentioned as a key organizational feature. The last goal was to separate the diagnostic specification, which is performed by diagnostic engineers, and reflects the diagnostic

knowledge, from the data specification and retrieval that is a programming task.

Paradigm Independent Design Concepts

GCESS was designed to implement multiple symptom driven paradigms. The design envisioned an abstract level of a symptom driven paradigm, which could be specialized or instantiated to define a particular diagnostic paradigm, which would then be incorporated into GCESS. The design also envisioned a general implementation strategy that used the schema and rule system of the underlying tool.

The GCESS design abstracted the key elements of generic symptom driven paradigms: *symptoms*, *measurements* (data), *diagnostic steps*, *diagnostic status*, *report generation*. A control mechanism was established called the *agenda*. Diagnostic steps would be *scheduled* by placing them on the agenda. The *diagnostic status* would contain the dynamic state of the diagnosis after each step that would be visible to the user in a status screen. A typical status screen for the Fault Tree paradigm (to be discussed below) is illustrated in Figure 1.

GCESS: motor diagnostics	
File Edit View Commands Options Help	
Unit Id: MOTOR UNIT ID #12RTS7	
Current Symptom: 3 of 3	
Name: TEMP_2	Type: PDA
Value: 101.2	Source: DS5 Time: 123
Last Fault Class	
Name: LUBRICANT_PRESSURE_LOW	
Status: SUCCEEDED	
Repair Recommendations	Test Recommendations
REPAIR_MOTOR_CIRCUI_1 REPAIR_MOTOR_CIRCUI_2 REPAIR_LUBE_LINE_2	
Notes	
No Initial Fault Class: Symptom: 1 PDA DS5 H1 No Recommendation for Symptom: 1 PDA H1 DS5	
Function Keys: F1=HELP F4=Run F5=Step F8=Halt	

Figure 1: Status Screen for Fault Tree Paradigm

A symptom driven diagnostic paradigm was assumed to be a specialization of the following generic algorithm which can be expressed in pseudo code as

```
while (there are symptoms remaining)
  begin
    select-next-symptom()
    schedule-first-diagnostic-step()
    while (agenda_not_empty)
      begin
        select-and-do-diagnostic-step()
        modify-and-display-status()
      end
    end
  end
end
generate-report()
```

Two important features of this description are that it is both vacuous (devoid of content) and procedural in nature. A specific *diagnostic paradigm*, is a particular specialization of the generic procedure. The individual paradigm defines conceptually how symptoms and diagnostic steps are scheduled, and how diagnostic steps are performed. This, in essence, defines a user model of diagnosis and gives content to the specific diagnostic paradigm. As a conceptual description, a paradigm does not specify how operations are accomplished at the implementation level.

The GCESS design assumes that a paradigm will define *diagnostic knowledge objects*, at the conceptual level, which will be used to schedule and implement diagnostic steps. It further assumes that these objects will possess a declarative form, that can be easily defined and manipulated by the engineer. It assumes they will be objects, in the sense that they can be created, modified, destroyed, edited, browsed, stored and retrieved by the user.

In the paradigms we implemented, the diagnostic knowledge objects are conceptualized as *diagnostic rules* (as opposed to ART-IM rules at the implementation level) which can be represented as *if then* statements

A particular *diagnostic application*, such as IMU diagnostics, will consist of a knowledge base of diagnostic knowledge objects, which direct a specific diagnostic paradigm to diagnose a component, together with a *data server* that will supply the symptoms and data to be used for the diagnosis.

An example of a diagnostic rule is an instance of:

if a particular signature is satisfied,

```
then
  recommend repairing the associated
  fault(s) that cause(s) it
```

This type of rule was the basis of the existing IMU software.

GCESS Implementation Strategy

The GCESS design envisions a generic implementation strategy for implementing a diagnostic paradigm.

The technical approach of a paradigm relies upon the schema, fact and rule system of the underlying tool. Symptoms, the diagnostic status, and the agenda are represented as schemas. A particular paradigm may add an additional status schema slot or define additional agenda item fields.

Diagnostic knowledge objects (diagnostic rules) are also represented as schemas. This gives them a declarative syntax and facilitates both forms-based editing and browsing, as well as object storage and retrieval.

While diagnostic rules at the conceptual level are represented as schemas, ART-IM rules at the implementation level are used to schedule and implement diagnostic steps.

These meta ART-IM rules can pattern match on the stylized conceptual diagnostic rules to select and schedule steps.

3. Developing and Implementing a Specific Diagnostic Paradigm

GCESS is designed to incorporate multiple diagnostic paradigms. In order to add a paradigm to GCESS, the shell must be extended as described in this section. This level of modification is extremely infrequent and is not expected to be performed by engineers or contractors.

Developing a diagnostic paradigm is a knowledge engineering task that defines a cognitive model of diagnosis that is close to the one actually used by group of engineers and technicians to diagnose a type of component. The model should be based upon rules or objects that are clear to the engineers and can be easily defined and understood in a declarative fashion. The paradigm should use or process these rules in a well defined manner that engineers and technicians can understand, so that they can manipulate the rules, and develop and maintain diagnostic systems using the paradigm. An example

of such a paradigm is the Fault Tree paradigm described below.

In order to implement a paradigm in GCESS, it is necessary to develop:

- An ART-IM schema representation of the diagnostic knowledge objects.
- Specializations of the agenda item and any additional status slots.
- An ART-IM rule set (group of production rules) to perform symptom and diagnostic step scheduling and to execute diagnostic steps.
- User Interface editors and browsers for maintenance of diagnostic knowledge object.
- A user Interface status screen and top level menu to invoke editors and browsers.
- Functions to save and restore the state of the current diagnosis.

Implementation of a particular diagnostic paradigm consists of specifying the schema representation of the diagnostic knowledge objects, and then developing the rules and functions necessary for symptom and diagnostic step scheduling, and executing each diagnostic step. Scheduling is performed by asserting and retracting from the agenda. An agenda item has a flexible format but must always include an (initial) paradigm identification tag.

Applications with mixed paradigms were considered but not implemented. Technically, this would require each paradigm to include a paradigm switching primitive, so that the UI display the correct paradigm specific status and menu. This is very simple to implement. However, mixed paradigms were not implemented because they did not seem to add any practical benefit to the two actual paradigms we developed.

A specific application for a paradigm will consist of a knowledge base containing the specific diagnostic knowledge objects required to perform the diagnosis within the paradigm. The knowledge base can be stored and loaded in textual form in a development environment. ART-IM provides the capability necessary to compile these objects into an executable form, and to construct a deployed application (executable image). Paradigm applications also specify the name of a data server that will supply the symptoms and data.

GCESS consists of the aggregate of the various paradigm rule sets, and diagnostic knowledge object editors and browsers, status screens and menus. Each diagnostic knowledge base contains a paradigm

identification tag. When a knowledge base is loaded, this tag can use to select the associated user interface menu and status screen.

Developing and using a specific diagnostic application consists of three steps:

- Develop the Diagnostic KB.
- Generate the Deployed Application Executable Image.
- Perform Diagnosis.

GCESS supports opening the diagnostic knowledge base, editing its objects, and saving the knowledge base, as well as performing diagnosis in the development mode to allow the engineer to develop the KB.

GCESS will generate C files and a make script to compile the knowledge base into a separate executable image.

GCESS performs the identical diagnosis in the development mode and the deployed image. It allows the diagnostic state to be saved after any state, and restored to resume diagnosis. The deployed image provides browsers but no editing facilitates.

4. Fault Tree Paradigm

The *Fault Tree paradigm* was the first paradigm incorporated into GCESS. This paradigm models a diagnostic procedure of finer and finer fault isolation.

In its simplest form it resembles a left to right depth first search tree, each node of which is called a fault class, and represents the class of terminal nodes (faults) which can be reached from it.

Figure 2 is a simple fault tree that represents a diagnostic procedure that says that if a motor is hot, we should check to see if the current is high, if the lubricant is low, and if the lubricant line is broken in that order.

The fault tree paradigm uses two basic types of diagnostic knowledge objects to control step selection. The *symptom/initial fault class map*, is a collection of diagnostic rules that selects the initial fault class to explore based upon the symptom features. It schedules the initial fault class on the agenda. For example, the map may indicate that if symptom name is "Motor_Temp_High" the Motor_Hot fault class should be scheduled. These diagnostic rules are used to implement the abstract schedule-first-diagnostic-item-on-agenda function.

The *agenda items* represent the fault classes that are scheduled to be explored.

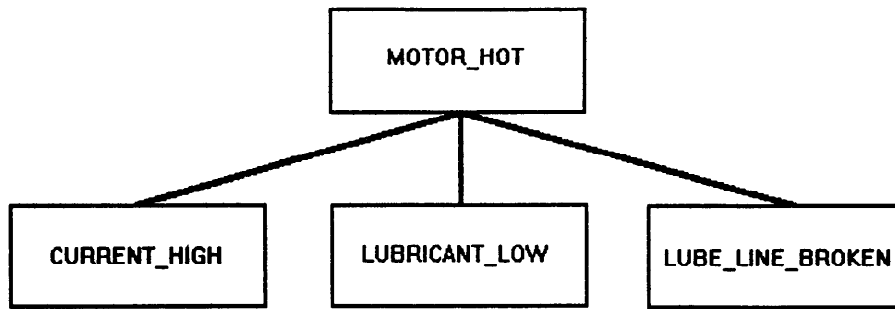


Figure 2: Simple Fault Tree Graph

The Lube_Line_Broken fault class above might represent the rule:

```

if Lube_Line_Pressure < 0.2
  then recommend Repair
    Broken_Lube_Line
  
```

Fault Class Types

Each fault class is also a diagnostic rule, which contains an *if* part and a *then* part.

The Motor_Hot fault class above might represent the rule:

```

if Motor_Temp > 110
  then
    explore Current_High,
    Lubricant_Low, Lube_Line_Broken
  
```

The fault tree paradigm defines four subtypes of fault classes. They are distinguished by their *then* or *action* types. Two subtypes indicate different ways of exploring other fault classes (which are represented graphically as children). The other two make different types of recommendations (repairs and tests).

The exploration types are called *ONE-OF* and *EACH-OF* actions. A *ONE-OF* fault class explores its child fault classes in left to right order until one of them results in a repair recommendation. It is roughly equivalent to saying "try this, and if that doesn't work then try this, etc." *ONE-OF* fault classes assume it is highly unlikely that more than one of the faults they lead to will occur simultaneously.

An *EACH-OF* fault class also explores its child fault classes in left to right order, but the exploration does not stop when one of them leads to a repair recommendation. It always explores all its children. It provides a way of isolating multiple faults. This may be used to check separate subsystems, or for multiple faults within the same subsystem.

ONE-OF and *EACH-OF* fault classes are represented graphically as nodes with arcs to children. The fault class contains not only the node, but also, in a sense, the arcs.

The two types of recommendation fault classes recommend repairs and tests respectively. All recommendations are accumulated and reported as the result of the diagnostics.

Repair recommendations indicate a possible fault and a repair procedure for the fault. Test recommendations indicate that further tests may be necessary to isolate the fault.

When a repair recommendation is encountered, all fault classes that led to its exploration are notified. This affects the *ONE-OF* fault classes that will not explore any other children. Test recommendations do not inform other fault classes. Using *ONE-OF* logic at a node, a repair recommendation (isolating a fault) will inhibit the search for further faults from that node, but a test recommendation (which does not isolate a fault) will not inhibit searching for faults.

Since *ONE-OF* and *EACH-OF* are specified by the engineer for each fault class, he has great control about where to search for multiple faults and where not to do so.

Fault Class Conditions

Each fault class also contains an *if* part consisting of a (possibly empty) set of conditions. These conditions must be satisfied for the fault class and its sub tree to be explored. For example, the Motor_Hot fault class might be a sub tree of a more general motor problem tree and contain the condition that the motor temperature is greater than 110°. In that event, the motor hot diagnostics would not be explored unless the temperature indicated that this was a possible problem. Initial fault classes for symptoms generally have no conditions, since they are the starting point for symptom diagnosis.

Unlike a decision tree, the conditions are not used to determine which branch should be taken, but rather whether the sub tree should be explored at all. When viewed graphically, the conditions are used to

isolate the fault to a set or *class* of faults reachable from the node. In reality, the fault tree is really a search tree representation of a diagnostic procedure.

The conditions are organized into *required* and *optional conditions*. All required conditions must be satisfied, but only some of the optional conditions may be needed for the *if* part to be satisfied. The engineer can assign a condition threshold, and weights associated with the group of all required conditions, and with each optional condition. Once the required conditions are satisfied, the optional conditions are tested until the threshold is exceeded. By adjusting the weights and threshold the engineer might indicate, for example, that two required conditions must be true and at least three of the five optional conditions must be true.

Edit Fault Class

Fault Class: MOTOR_CURRENT_HIGH

Type	Clause	(Wt)
Req	CUR > 4.2	
Req	VOLTAGE < 8.1	
Opt	TEMP > 98	30
Opt	MOTOR_SPEED > 2000	30

Req Wt:

Threshold:

Action Type: REPAIR-RECOMMENDATION

Severity:

Actions

100 REPAIR_MOTOR_CIRCUIT

Parents:

Figure 3: Fault Class Top Level Editor Screen

The conditions themselves are Boolean expressions that generally involve measurements (often sensor data) and ranges. This allows a simple mathematical declarative syntax with a simple mechanism for practical fuzziness (optional conditions). There is a well defined condition language including standard mathematical functions.

The base ART-IM language may be used to define additional functions. Conditions use a well defined mathematical expression syntax like FORTRAN,

The engineer specifies fault class as *condition action* rules as illustrated by the Figure 3, which shows the top level fault class editor screen for a particular fault class.

Assumptions and Replicated Components

When a ONE-OF or EACH-OF fault class schedules its children to be explored, it may also specify a list of *assumptions* to be used by each child. These are specified on a child by child basis and may differ among children. Each assumption assigns a value to a

specific variable. This mechanism is provided to parameterize the logic and allow the same logic to be applied to multiple instances of replicated objects. The assumptions are accumulated dynamically, so that each fault class inherits the assumptions of all fault classes that led to its exploration. Measurement names and recommendation names can be "macro expanded" using the assumptions and a simple name concatenation form specified by the engineer. For example, the fault class Check_Lube_Line

```
if Lube_Line_Pressure < 0.2
  then recommend Repair
      Broken_Lube_Line ,
```

might contain the parameterized measurement name "Lube_Line_Pressure" and parameterized recommendation name "Repair Broken_Lube_Line" whose macro expansions depend on the value of the assumption variable Lube_Line_Num.

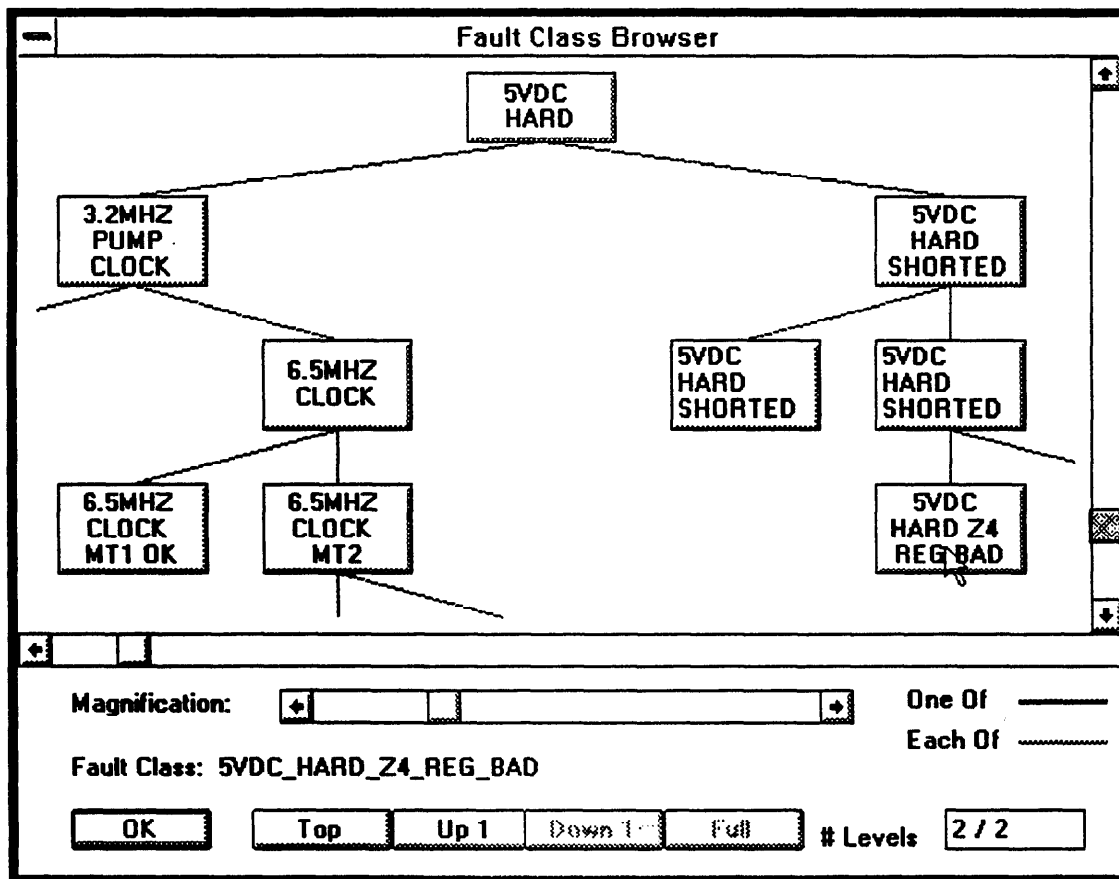


Figure 4: Graphic Browser

A more general fault class, `Check_All_Lube_Lines` might specify the same child `Check_Lube_Line` three times but with the different values of `Lube_Line_Num` of 1, 2 and 3. This would apply the same logic to the three separate lube lines, and explore the same fault class three times, but using different assumptions.

Fault Class Recommendations

Fault class recommendations may contain a recommendation name and text, as well as explanation, and links to hyper text files (such as a Theory of Operations manual) and graphic images. These are supplied by the engineer in the fault class. Recommendations also contain the symptom that led to the recommendation, the list of current assumptions, and the fault classes that led to the recommendation being made. Repair recommendations are reviewed by technicians. They have the ability to accept or reject them, and the information necessary to make that decision. In this sense, the diagnostic application is an advisor. This kind of advisor allows the technician to both criticize its results and suggest changes that can be rapidly incorporated. It is easily viewed as a tool, rather than as an adversary or inconvenience imposed from on high. This produces a great advantage in gaining organizational acceptance. In a sense it can co-opt its strongest critics, by turning them into participants in the development of diagnostics.

Fault Tree as a Graph

The term "Fault Tree" is somewhat of a misnomer. It need not be a "tree" in the topological sense. It is actually a directed graph that may contain both cycles and disconnected components. It is typical, though not necessary, for individual symptoms to map to disconnected graph components. The paradigm will not explore the same fault class twice for the same symptom under the same set of assumptions, but static cycles in the graph are permitted.

The Fault Tree Paradigm UI

The fault tree paradigm contains a status screen that displays status after each step, editors and browsers for the initial fault class map that maps symptoms to root fault classes, and the fault classes themselves. It also provides a graphical browser to view the network and edit fault classes. The browser is illustrated in Figure 4. Fault class objects contain a great deal of information and there are a variety of screens used to support editing and browsing fault class elements. In all, there are around 50 dialog

boxes and screens used by the fault class paradigm. However, many screens have editor and browser formats that are nearly identical.

The Importance of Explanation

A primary feature of the fault tree paradigm is its ability to reduce the cognitive distance between the way the diagnostic software diagnoses the data, and the way in which the engineers diagnose the data. The technicians can accept the diagnosis because they can read and understand the fault class diagnostic rules and explanations.

At the beginning of this paper, it was argued that diagnostic rules are often derived from inverting deductions in an informal mental causal model. Fault classes express rules in a crisp mathematical manner. However, they may not fully express physical and engineering principles, or actual experience represented by the mathematical forms. For this reason, the fault class rules may be annotated with a detailed explanation supporting the rule. The use of hypertext and graphics links, allow the explanation to refer to Principles of Operation manual and diagrams.

In practice, these executable diagnostic rules, annotated by supporting explanation, provide a practical repository of diagnostic knowledge and experience.

5. Separation of Data from Knowledge

GCESS applications can deal with very large amounts of data. This data is fetched on demand. The data is modeled as a *measurement* that is identified by a name, data source and time. Each GCESS knowledge base must specify the name of a *data server*.

The data server is responsible for implementing the retrieval of symptoms and measurements, and performing application specific data processing. Data servers may use flat files, data bases or any means they want. GCESS specifies two generic functions to get symptoms and measurements that must be provided by a data server.

Data servers are implemented as Windows *Dynamic Link Libraries* (DLLs). These are separate executables that are developed on a per application basis by programming staff. This ensures separation of the specification of the diagnostics by the engineers using GCESS, from the programming tasks associated with data retrieval and processing. The DLL mechanism that resembles a remote procedure call allows different applications to link to different versions of the same function.

6. Additional Diagnostic Paradigms

Several diagnostic paradigms were considered and rejected for use in GCESS. These included model based reasoning (Davis et al., 1982, Hallanti, Stafanini & Tomada 1989), use of traditional fuzzy logic (Zadeh 1965), and a paradigm developed internally at Inference for the integration of separately developed component diagnostics. The reasons they were rejected had to do with the fact that they did not correspond to the way the actual engineers conceptualized diagnosis, and consequently could not be maintained or serve as a repository of knowledge.

Some thought was given to using symptoms to isolate the fault to a subsystem, which could then be analyzed by model based reasoning. However, it was unclear that this was actually the case. Furthermore, this could result in merely transferring the software maintenance problems to the process of modeling a complex physical component.

Fuzzy set membership functions per se did not seem to reflect the technicians point of view. Recommendations generally list the measurements used to arrive at them. Technicians can view these values and can query for others to determine if the conditions are really satisfied. For example, if a condition of the form "temp > 100" were satisfied and the value of temp were 100.0001, the technician might not accept the recommendation. They definitely wanted to be "in the loop" in these sorts of decisions, and wanted the leeway to exercise their own judgment as opposed to relying on automatic (fuzzy) techniques.

An additional diagnostic paradigm for Circuit Analysis was incorporated into GCESS. This paradigm was developed by Rockwell for diagnosis of test stations themselves. This paradigm was eventually eliminated after the delivery of the Rockwell prototype application, presumably for lack of funding to continue the development of this type of diagnostic application (i.e., test station diagnostics). It did, however, validate the concept of a multiple paradigm shell.

The Rockwell paradigm was related to a model based reasoning process. It required a very large "hardware model" that needed to be separately maintained and defined the diagnostics to a great extent. This led to real concerns in terms of maintenance, that we described above.

Case based reasoning methods (Hammond & Hurwitz 1988) were considered but not included for

the applications considered. The feeling was that if a diagnostic case were successfully analyzed and understood, it could be expressed as a fault class. Use of a case base search of similar problems was thought to be of limited utility, and as a method of last resort. We still have some interest in incorporating a case based paradigm to broaden the scope of the tool, provided additional funding can be obtained.

Neural net methods were also considered to support pattern recognition and discrimination tasks but rejected for practical and economic reasons. Many analysts compare the actual plot of data parameters with nominal plots of normal and faulty components. In the IMU case, there were hundreds of such plots that might be considered. Although, using neural net discrimination was attractive theoretically, it would involve the preparation of a very large amount of training data, and interfacing with other data analysis programs. In the more general shell setting, it meant that an engineer would have to train a net for each discrimination, which presented ease of use problems. It was far more practical to have the engineer indicate the patterns and intent, and have the technician make the discrimination when necessary. Humans are obviously very good at this type of pattern recognition, and the Windows environment supports running other data analysis and plotting programs simultaneously.

7. Fielded Applications

Several major applications have been developed using GCESS. The IMU ADS application, developed by Northrop, uses approximately 1400 fault classes and was first deployed in June of 1992. It takes under 2 minutes to run and dramatically increases the fault coverage over the previous diagnostic software (ATP). Maintenance technicians are satisfied that the logic is correct. Creating or modifying a fault class is essentially a text processing operation, and a new deployment version can be created in under an hour. There are four copies of the IMU diagnostics currently deployed at three separate sites.

This application was created by starting with the specifications for the original diagnostic program, and conducting extensive knowledge engineering interviews to incorporate current diagnostic knowledge. The interviews were conducted by a senior Northrop systems engineer, who was the project leader and responsible for creating the diagnostic rules. He first had to create the overall diagnostic flow, and grouping of problems and sub

problems. He structured the overall problem so that he was confident in the diagnostic coverage of the many possible symptoms. The diagnostic problems were grouped in about twenty broad categories. The diagnostic rules cover a variety of diagnostic tests that are employed and use several different files of time averaged data.

The interviews were conducted over many months and were with other members of the engineering staff who were expert analysts of the various subsystems. While the interview process was progressing the lead systems engineer wrote the diagnostic rules that form the diagnostic application. He did not have to rely on programming personnel or professional knowledge engineers, and had complete control over the diagnostics. He included the engineering explanations as part of rules. The explanation is quite extensive.

A second IMU GCESS diagnostic system (FDS) was constructed to analyze IMU field data (as opposed to ATE data) at the repair facility. Field data sometimes accompanies the components that arrive for repair. This replaced a previously deployed diagnostic program (FTP), and was constructed by Northrop engineers.

A Stable Member diagnostic application has been developed by the repair facility engineers. It uses some of IMU rules and has a much larger interactive components. It is on the order of 800 rules.

Rockwell International developed a Field Data Advisor (FDA) which examines fleet status and uses GCESS as the embedded diagnostic component. Assessment experts track a variety of parameters and anomalies at individual sites, as well as parameters for the entire fleet. They are concerned with the status at any given moment and how it evolves over time. FDA is a tool to automate parts of the analysis. FDA was tested and deployed at the end of 1992. Three copies of FDA are currently deployed at three separate sites.

The GCESS portion of FDA was constructed with the involvement of the fleet assessment experts, who are themselves engineers. It uses approximately 850 fault classes. The rules analyze data produced by signal processors to diagnose on going problems and trends. They both consult and maintain an historical track list of the processed parameters and problems.

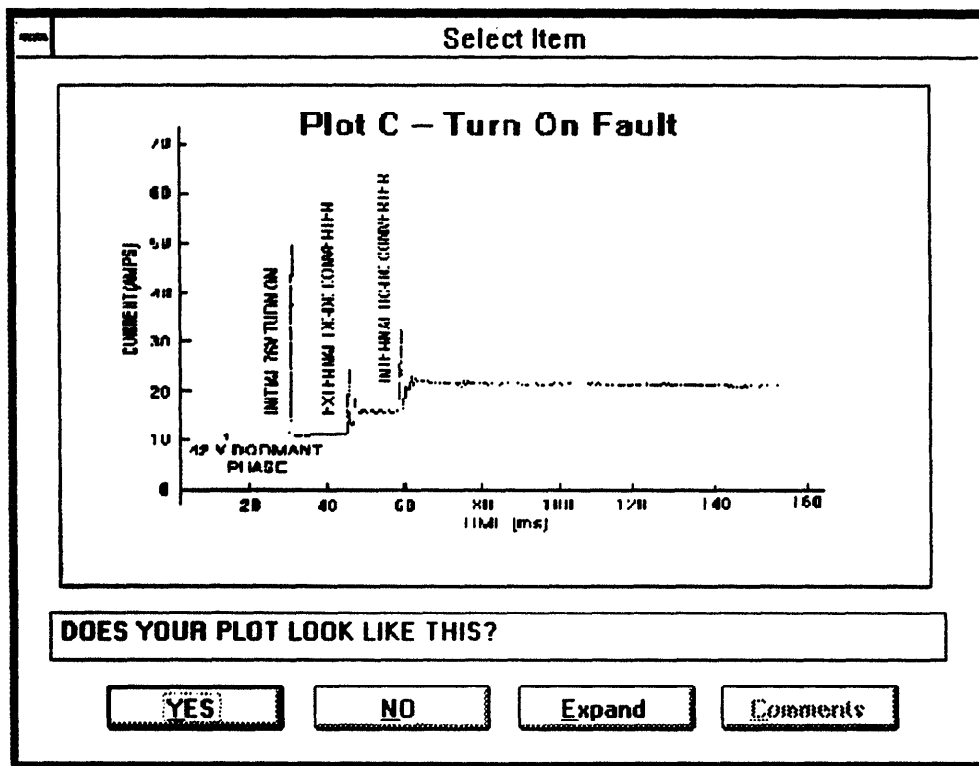


Figure 5: Interactive Question Displaying Graphic Image File

A Spectrum Analyzer diagnostic application was developed at the repair depot. This application is smaller in size and is highly interactive. It contains around 200 rules.

Although GCESS was originally developed to primarily support diagnosis based upon the results of prior extensive testing, it can be used in an interactive fashion as well. It contains functions to ask yes or no questions, list selection questions and questions for values. These question functions display graphic image files. This means it easy to indicate a question like "does the oscilloscope pattern look like this?", and attach a graphics image of the nominal pattern, as illustrated in the Figure 5. These are used in place of a more elegant pattern analysis as discussed above.

There are currently twelve copies of GCESS which are used at 6 sites, in order to run five distinct applications. Northrop, Rockwell and TRW have copies of GCESS. The Air Force has copies of GCESS at two of its logistics sites.

8. Success and Benefits

Success

GCESS has been successful in developing diagnostic applications that are actually being used by AGMC repair technicians to diagnose missile components. Five separate deployed applications have been built.

The IMU diagnostic applications, unlike their conventional software counterparts, are in active use. We consider both IMU applications to be major successes, because they perform well and achieve their goals of producing accurate diagnosis, and valuable explanations.

One interesting point is that much of the diagnostic knowledge from the original applications was used in building the new ones. The modularization of this knowledge, in fault class rules, and the ease in which it can be specified and modified *by engineers alone*, are major factors of the success of these applications, and indicates the success of the shell itself.

While the IMU applications were seen to redress the problems with the existing software, they also addressed the deeper problems of creating a repository of diagnostic knowledge. This knowledge, embodied in fault class rules, has been successfully transferred from Northrop to the Air Force. In this sense both the applications and the shell have been successful. This is a major benefit, although it is difficult to quantify.

After about a half day's informal training in GCESS at the repair facility, the repair facility

engineers began developing and deploying several diagnostic systems of their own, without extensive training or defense contractor involvement. This demonstrated their confidence in the system and its ability to serve as a repository of their diagnostic knowledge. They obviously felt they could express and maintain this knowledge using the GCESS formalism and that it would be accepted by technicians because it could be understood. We consider this a measure of the success of the shell in achieving its objectives.

Cost Benefit Analysis

Guidance and control components are produced in small numbers and require expert engineering diagnosticians at both the contractor and repair facilities. The maintenance of the components is extremely costly, especially on a per unit basis. The development of new diagnostic support was viewed as an operational necessity, and no formal cost benefit analysis was performed as part of the justification for this work.

However, informal analysis indicates that the *combined* cost of developing the diagnostic shell and the first major IMU application was competitive with the expected cost of development of a new conventional software version alone. From a return on investment point of view, the investment in the shell is returned in the reduced cost of its first major application. This is an extremely favorable break even point, and can be used in an informal cost benefit analysis of the shell, as opposed to the specific applications.

Since the cost of the shell is paid for by the first application, its financial benefits are realized by subsequent applications and maintenance phases.

The reductions in costs of subsequent applications, such as the IMU field data application (FDS), occur because the work necessary to construct diagnostic rules *must be done anyway* as part of the conventional software life cycle. The engineers must analyze the problem and come up with diagnostic procedures in order to write the conventional software specifications. The shell approach saves the cost of implementing the specification in procedural code. This is a major financial benefit of the shell.

The second major financial benefit is that it reduces the cost of software maintenance dramatically.

As previously mentioned, the actual applications are justified on the basis of operational necessity rather than cost/benefit.

9. Innovative Aspects

The use of AI diagnostics in the real life diagnosis of guidance components is new in the Air Force missile realm. It represents a dramatically different approach than the traditional highly controlled conventional diagnostic software. It has shown that it is possible to provide diagnostic software that is maintainable by the engineers and keeps abreast of the diagnostic life cycle.

The development of an architecture for hosting multiple diagnostic paradigms is another of the innovations of GCESS.

Acknowledgments

The development of GCESS and its applications required the assistance of many people within the Peacekeeper community. We would like to acknowledge the help of Nevin Colwell of TRW, Chris Grau of Inference, Bill Hutton of Inference, Clay Pinkerton of Rockwell, Dave Reed of Northrop, Diana Rice of TRW, Jeff Royle of Northrop and Capt. Terry Sirbough of the U.S. Air Force.

References

Davis, R., et al. Diagnosis based on description of structure and function. Proceedings of the National Conference on Artificial Intelligence Applications, AAAI, 1982.

Grau, G. , Holtzman, P. GCESS: An Expert Diagnostic Shell, Proceedings Test Engineering Conference, 1991.

Hallanti, M.,Stafanini, A. Tomada, L. ODES: a diagnostic system based on qualitative modeling techniques. Proceedings of the Fifth Conference on Artificial Intelligence Applications, IEEE, 1989.

Hammond, K. , Hurwitz, N. Extracting Diagnostic Features from Explanations. Proceedings of a Workshop on Case Based Reasoning, DARPA, 1988.

Zadeh, L.A. , Fuzzy Sets, Information and Control 8, 1965.