System Complexity and Integrated Diagnostics

WILLIAM R. SIMPSON
JOHN W. SHEPPARD
Arinc Research Corp.

In part 1 of a series, the authors give an overview of a complete approach to integrated diagnostics. The approach is centered around an information-flow model and incorporates techniques from information fusion and artificial intelligence to guide analyses. The authors explain how to analyze testability, evaluate fault diagnosis, and create maintenance aids. Each subsequent article in the series will elaborate on the parts of this approach.
(See pp. 18-19 of this issue for an overview of the complete series.)

The complexity of modern systems is putting new demands on system maintenance. Every system, whether airplane, radio, or computer, has a mission to perform. The primary goal of system maintenance is to keep the system available for that mission. When the system fails, the job of maintenance is to diagnose and repair the failure as rapidly as possible to return the system to correct operation. But diagnosing failures in complex systems requires analyzing system characteristics in great detail.

How do you reconcile the need for rapid repair with the need for in-depth analysis? Fault-tolerant systems approach the problem by limiting the need for diagnosis and repair, identifying failures as they occur on line, and reconfiguring the system to maintain functionality. Those in field maintenance have also tried to provide system-level diagnosis, incorporating ad hoc procedures based on field expertise, but this process is independent of design and manufacturing for the most part.

The general problem of diagnosis is extremely difficult. Optimizing a diagnostic sequence of tests is known to be an NP-complete problem, and deriving heuristics to improve practical diagnosis is a long-standing problem.[1-4] In attempting to address these problems, developers have created several tools to build efficient fault-isolation strategies using many approaches, including brute-force/exhaustive search, heuristic search, and entropy-directed search.

In this article, part 1 of a series, we describe a method of assessing and diagnosing testability that uses several well-established techniques from artificial intelligence. The method provides a way to describe the flow of diagnostic information through a system, and the resulting model serves as the knowledge base for several analysis tools. At Arinc, we have applied this method to assess and diagnose the testability of many systems over the last 10 years.

Integrated diagnosis

In the early 1980s, industry and government developed several initiatives to help keep pace with growing system and diagnostic complexity. From these programs and initiatives, groups

0740-7475/91/0009-016/$01.00 © 1991 IEEE

are now developing useful testing and diagnostic products, some of which are becoming well recognized in the automatic test community.[5-7]

Unfortunately, each of these initiatives treated only one aspect of the life-cycle testability problem or treated each aspect as a separate issue. None of the initiatives significantly addressed the underlying philosophy of integration or its impact. Instead, integrated diagnosis was treated as nothing more than file and data sharing.

As diagnosis and maintenance began to receive more attention, the focus shifted to methods that more truly reflected the nature of integrated diagnosis. In 1990, Keiner formally defined integrated diagnostics as[8]

"...a structured process which maximizes the effectiveness of diagnostics by integrating the individual diagnostic elements of testability, automatic testing, manual testing, training, maintenance aiding, and technical information."

This definition encompasses much more than file and data sharing. Here, the primary goal of integrated diagnosis is to optimize field-maintenance resources within the system's operational environment. Strategies include minimizing the mean time to isolate system faults, the mean time to repair systems, and the sparing requirements associated with systems. Training is also included as an area of concern. By reducing the need for specialized diagnostic skill, we can reduce the impact of losing experts and expertise and reduce the training costs.

The elements

Following Keiner's definition, the principal components of an integrated diagnostic system are tools that allow us to evaluate the design and develop diagnostic strategies. Design evaluation consists of analyzing the system in conjunction with its set of tests to determine the test set's specific diagnostic capability. On the basis of the results, the testability analyst may design new tests, eliminate or redesign old tests, or repackage the system to meet testability goals or confine potential testability problems to single, replaceable unit packages.

The process used to modify a design to meet a set of testability goals is called *design for testability*. Testability goals may be generic, such as those identified in Mil-Std-2165,[9] or they may arise from specific design criteria. These goals are usually set at the system or mission level.

Electronic maintenance aids make diagnosis more effective. Training programs need to cover the use of these aids.

The process used to attribute these goals to a specific subsystem or hardware element is called *testability allocation*.

Once the design team defines and determines the testability for a system, testability analysts from that team can develop diagnostic strategies or procedures to identify faults as they occur. This strategy development frequently involves the integration of automatic and manual testing. In avionics systems, for example, the built-in-test (BIT) system on an airplane may identify one of the avionic units as having failed. When the airplane returns to the hangar, the maintenance technician can pull the unit identified and test it with automatic test equipment to determine which card within the unit contains the fault. The ATE may also identify the card's failed component or components. An alternative is to have the technician manually test the unit to find the faulty component.

Electronic maintenance aids and on-line technical information make diagnosis more effective. These aids permit a technician prompt and easy access to the information needed to identify the fault and make the repair. Training programs for technicians need to cover the use of these aids and tools so that technicians can effectively, efficiently, and reliably repair and maintain the system as required.

When implementing a hierarchical, integrated maintenance architecture, we must consider how on-line and off-line test systems are integrated. BIT and performance monitoring/fault locating systems provide on-line monitoring of a system's performance and health state. As problems arise, these systems flag and report the problem so that technicians can take the appropriate action. Any actions taken and the results of tests are stored for use as input to the diagnostic process at the next level.

According to Keiner's definition, we must apply a carefully determined structure to the problem in a way that guarantees a complete and consistent architecture. A structured approach ensures that the problem is properly and appropriately represented. That is, the approach does the following:[9-11]

- covers hierarchical details
- is applicable at different maintenance levels
- includes information relevant to different technologies and interfaces
- represents system details to enable either on-line or off-line testing
- provides a mechanism for efficient analysis

Testability definitions

Keiner's definition includes testability as one of the elements in integrated diagnostics. Mil-Std-2165 defines testability as[9]

"...a design characteristic which allows the status (operable, inoperable, or degraded) of an item to be determined and the isolation of faults within the item to be performed in a timely and efficient manner."

We say that equipment has good testability when we can confidently and efficiently identify existing faults. Frequently identifying only the failed components or parts without removing good items establishes a high confidence level. Efficiency is optimizing the resources required, including staffing, labor hours, test equipment, and training.

The literature describes at least two types of testability. *Inherent testability* is a design characteristic that provides the potential ability to observe system behavior under test stimuli. The location, accessibility, and sophistication of tests that we may include in the system define its inherent testability.

Achieved testability, on the other hand, is a maintenance characteristic that provides the actual ability to observe system behavior under test stimuli. We measure achieved testability by the results of the diagnostic process, including the set of tests defined for the system, realized false alarms, ambiguities, and correct as well as incorrect diagnoses.

Achieved testability is the same as inherent testability if all diagnostic tests are available, no diagnostic tests are subject to false alarms, and the fault-isolation techniques fully exploit inherent testability. For this reason, inherent testability is the upper bound on achieved testability, while achieved testability has no practical lower bound except perhaps no testability.

Central to achieving maximum testability is the design of the test. *Test* here is "a signal, indication, or other observ-

MODEL-BASED DESIGN FOR TESTABILITY: ONE COMPANY'S APPROACH

The 1980s saw a shift from systems that emphasized design for performance to those that stressed design for field operations. With this shift has come an increased demand for systems that sustain a certain performance level throughout their life. Customers are demanding to know how maintainable systems will be in the field, almost before they ask about performance.

Clearly then, the old way of doing business is giving way to a more structured approach. Companies can no longer afford the luxury of designing to performance specifications and then supplying maintenance and diagnostic procedures as an afterthought. This afterthought, or nonintegrated, approach led to unsatisfactory conditions: Retest OK, or RTOK (pronounced ree-tock) rates were 40+ percent with field NFF (no-fault-found) rates of 50 percent, and false-alarm rates often exceeded valid detections.

Companies tried to combat these deficiencies in testability by providing detailed specifications, but results in the field diverged significantly from the ad hoc testability measures the designers had taken. In 1985, recognizing the problem, the US Department of Defense instituted Mil-Std-2165 to require a detailed analysis of testability issues during design.

Since then, a number of companies have developed approaches to comply with the standard. Many use a model-based approach to design for testability and diagnosis. This article is the first in a series reporting on work done by Arinc Research Corp. in developing a model for integrated diagnostics. The series, which documents 10 years of development and application, addresses the mathematical approaches taken in Arinc's System Testability and Maintenance Program, called STAMP, and Portable Interactive Troubleshooter, called Pointer.

Arinc's model-based approach, called information-flow modeling, differs from the traditional model-based approach because it does not use a physical model of the system during diagnosis. Instead, a limited form of intelligence is placed into the diagnostic process, taking into account any known information, such as built-in-test readings, operator reports, logistics history, and specific symptoms. The model represents the knowledge base of the system to be tested as the system's flow of diagnostic information. Consequently, the maintenance technician has maximum flexibility because the model can adapt to changing conditions during maintenance and is able to work around deficiencies in test equipment and other factors.

The first article in the series is an overview of the information-flow model's structure. Later articles will expand on this structure and related subjects.

The article on *assessing system testability* includes graphical representations, groupings, and multiple-conclusion mapping and describes in detail the processing of the model, including higher order representations and the

able event that may be a normal output of a system or be caused to happen."[12] This definition is based on the concept of *information fusion*, in which a test serves as information that we can apply to the diagnostic problem. We must then fuse the multiple sources of information to correctly diagnose the fault.

This definition of test is much broader than the typical, more restrictive definition of test as a stimulus-response pair indicating the behavior of the system. What a maintenance technician may see or hear, for example, qualifies as a test. Since most tests can have two outcomes—pass and fail—we assume that all tests are binary. In this way, we control combinatorial explosion by requiring a multiple-outcome test to be described as a set of binary-outcome tests.

As with testability, *diagnosis* often represents more than one concept, usually distinguishable by context. We are concerned here with three aspects of diagnosis, all of which apply whether we are considering on-line monitoring or off-line diagnosis. *Detection* refers to the ability of a test, a combination of tests, or a diagnostic strategy to identify that a failure in some system has occurred. This term is often associated with BIT and may actually be the primary design criterion for BIT.

Localization is the ability to say that a fault has been restricted to some subset of the possible causes. This also is associated with a combination of tests or a diagnostic strategy. Clearly all BIT that detects faults also localizes them to at least one of all possible faults. If the localization is sufficient to repair or reconfigure the system, we often refer to BIT as smart BIT. BIT, however, is not the only diagnostic technique that localizes faults. Often ATE and manual isolation use diagnostic strategies that localize the fault sufficiently to repair or reconfigure the system.

Isolation is the identification of the specific fault through some test, combination of tests, or diagnostic strategy. Isolation in this article is restricted to localization that is sufficient to repair a single unit at a specific maintenance level.

types of information that can be derived from these representations. The authors compute several testability characteristics and describe their use in assessing system testability.

The article on *using test resources* describes one of the classic problems arising from an ad hoc approach to design for testability: some portions of the system are overtested and other parts are undertested. The article gives procedures for analyzing the test set with the goal of providing the minimum testing to achieve all testability objectives—including incorporating multiple failures, compensating for false alarms, identifying inadequate test resources, and eliminating unnecessary test resources.

In an article on *static and dynamic fault isolation*, the authors describe fault isolation as a search problem in which selected tests prune the space of possible diagnoses. They review approaches to diagnostic search with emphasis on an approach that uses information theory. They also derive entropy-based approaches to choosing tests using a set of inference rules tailored to fault diagnosis. An interactive process that uses test-choice and inference algorithms allows diagnosis without precomputing fault trees. Details are given on how to incorporate machine learning into diagnosis.

Although inference has been applied successfully to many diagnostic problems, testing frequently yields uncertain or incomplete information. In an article on *fault isolation under uncertainty*, the authors describe how uncertainty may arise because of poorly defined tests, inaccurate instrumentation, or inadequate skill levels. Within an interactive environment, confidence in test data can be input to diagnostic reasoning. The authors' approach to reasoning with uncertain or incomplete information incorporates solutions to a number of problems, such as specifying certainty in test results, computing certainty values for possible diagnoses, and recommending a repair action.

All the analyses for diagnostics can be combined in one integrated approach to diagnosis and repair. In an article on *integrated diagnostic architectures*, the authors cover the basic elements of an integrated architecture, including design for testability, testability allocation, logistics support, built-in test, performance monitoring, embedded maintenance, automatic test equipment, electronic maintenance aids, logistics feedback, and training.

The series encompasses more than 10 years of work on developing these techniques and applying them to the testability and diagnosis of complex systems. All the areas described have been directly addressed in applications with successful results. To date, Arinc has not fully applied this integrated architecture, but several full applications are currently underway. Full diagnostic architecture integration may take considerable time but, ideally, will span the life cycle of the system.

The model

Our model-based approach incorporates techniques from information fusion and artificial intelligence to guide analysis. The model represents the problem to be solved as information flow. Tests provide information. Diagnostic inference combines information from multiple tests using symbolic logic and pattern recognition.

Primitive elements

Because the model uses information fusion, testability analysts have to consider the information gained from performing a test as they develop the model. The analyst begins by specifying the primitive elements and then proceeds to a description of the logical relationships and groupings of these elements.

The primitive elements of the model are the tests and fault-isolation conclusions, which are based directly on information fusion. Tests correspond to the information sources, while fault-isolation conclusions correspond to the set of conclusions that can be drawn after running tests. A test is any source of information available that the analyst can use to discern a fault-isolation conclusion. A fault-isolation conclusion is any element that we can isolate within the model. Thus, a conclusion is often a failure mode of some component or functional unit within the system.

The model also includes three special primitive elements: testable input, untestable input, and No Fault. The inputs represent information entering the system that may have a direct bearing on the health of the system. A testable input is a conclusion corresponding to an external stimulus combined with a test that examines the validity of that stimulus. If we have an input that cannot be examined for validity, that element is called an untestable input. Finally, the model includes a special conclusion corresponding to the condition that the test set found no fault. No Fault, also referred to as RTOK (for retest okay), provides us with a closed-set formulation that includes anything not directly accounted for.

The analyst organizes conclusions according to the required repair level. Conclusions include line-replaceable units (LRUs) if the need is at the organizational level, shop-replaceable units (SRUs) if it is at the intermediate level, or components if it is at the depot level. Further, the analyst can develop models that cross levels. That is, in a single model, a conclusion may be a subsystem, an LRU, an SRU, a component, or a failure mode, depending on what is appropriate.

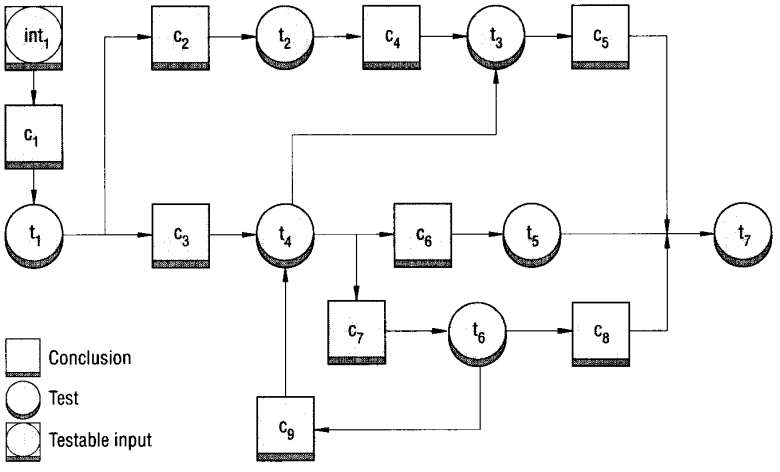

Figure 1. Example of a dependency graph.

Dependency relationships

After specifying the primitive elements, the next step is to determine the logical relationships among the tests and between the tests and the conclusions. To determine logical relationships, the analyst considers the following for each test:

1. inferences drawn from a test failing
2. inferences drawn from a test passing

In the initial stages of developing a model, the first issue is more important. The modeler is interested in listing conclusions that, corresponding to a failure, would explain the current test failure. The modeler is also interested in listing tests that, should they fail, would cause the current test to fail. If such tests do exist, we say that the current test *depends* on tests and conclusions that may cause it to fail. That is, a dependency relationship exists. The second question is important in determining the type of test (for example, whether the information provided is symmetric).

We represent dependency relationships in the model as a directed graph in which tests and conclusions are nodes and dependencies are edges. Figure 1 illustrates such a graph. If test t_2 depends on test t_1, then either an edge is drawn from t_1 to t_2, or a path exists from t_1 to t_2 in which all other nodes on the path are conclusions. We can interpret the dependency graph as "If t_1 fails, then t_2 will also fail."

Figure 1 shows edges from tests to

conclusions. This representation is an artifact of the way analysts typically create dependency models. In general, we cannot say that if a test fails, then a conclusion will fail, unless test performance causes a system component to fail.

As an alternative, we can consider a logical representation of dependency, shown in Figure 2 as an edge from t_2 to t_1. We interpret this representation as "If t_2 passes, then t_1 must also pass." Clearly, we can draw a similar graph for a test that depends on a conclusion. The graph would have a path that terminates at a conclusion or input. In this approach, conclusions never depend on tests.

The model differentiates between two orders of dependence. A *first-order* dependency corresponds to a single edge in the flow graph. The dependence of t_2 on t_1 and the dependence of t_3 on t_2 in Figure 2 are first-order dependencies. A *higher order* dependency is determined by the transitive property of implication applied to dependencies in the flow graph. That is, it corresponds to a path in the graph that includes at least one additional test. The dependence of t_3 on t_1 in Figure 2 is a higher order dependency.

Test variants

The model allows the analyst to specify several types of tests, including symmetric, asymmetric, cross-linked, and conditional. The basic test form is the symmetric test, which provides complementary information given a pass outcome and a fail outcome. Let **A** be the set of candidate conclusions when a test fails, and **B** be the set of conclusions that are no longer candidates when the test passes. If $\mathbf{A} \equiv \mathbf{B}$, then the corresponding test is symmetric.

To determine inferences drawn from a test passing the analyst lists the conclusions that have not failed and the tests that should also pass. If this list is the same as the list of tests and conclusions that remain after eliminating inferences drawn from a test failing, then the test is symmetric.

If a test is not symmetric (conditions for symmetry do not hold), then it is asymmetric. An example of an asymmetric test is a warning light on a panel. If the warning light is on, the technician learns that an alarm condition has occurred. On the other hand, if the light is not on, the technician learns very little because the bulb or voltage to the bulb may be bad.

An asymmetric test can be one of three types. In a *positive-inference* asymmetric test, all elements in the test's dependency list will pass if the test passes, but no information is gained if the test fails. The *negative-inference* asymmetric test is mathematically similar to the positive-inference test. If a negative-inference test fails, then the elements in the test's dependency list are the candidates and all tests dependent on the negative-inference test will fail. But if the test passes, no information is gained.

The third type of asymmetric test is the *fully asymmetric* test. This test combines the characteristics of the positive-inference test and the negative-inference test. In fact, it is actually a positive-inference "subtest" and a negative-inference "subtest" linked. If the fully asymmetric test passes, then the dependencies specified by the positive-inference subtest are considered good elements and removed from consideration. On the other hand, if the fully asymmetric test fails, then the dependencies specified by the negative-inference subtest are considered the set of failure candidates, and all other elements are removed from consideration.

Related to the fully asymmetric test is the *cross-linked* test. Cross-linkages occur when the outcome of one test implies the opposite outcome of another test. For example, if the warning light described earlier is on (the test fails), then a press-to-test of the bulb circuit will pass. The analyst can link any two tests, t_1 and t_2, in one of three ways:

1. The passing of t_1 implies that t_2 fails.
2. The failing of t_1 implies that t_2 passes.
3. Both 1 and 2 are true.

The last test variant, a *conditional* test, is a more general form of asymmetric test. In a conditional test, a system state or mode, such as user inputs, scale settings, switches, or operational modes, determines the list of dependencies. The model represents conditional tests as

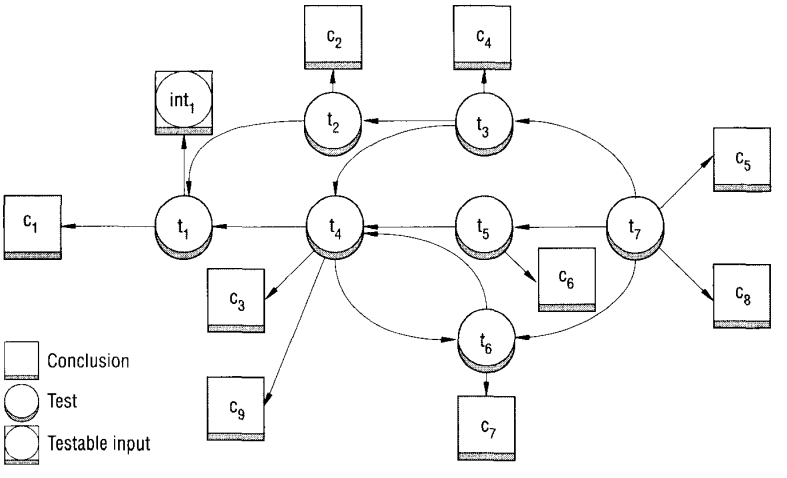

Figure 2. *Logic diagram for the dependency graph in Figure 1.*

copies of some general test. Each copy has a dependency list appropriate to its condition. When the test is specified, the system is placed in the condition required for that test. Of course, during diagnosis, the technician may wish to control or restrict the mode.

Diagnosing conclusions

In diagnosing a failure, the inference system reports either a single conclusion or many conclusions. In *single-conclusion* diagnosis, it reports a fault after identifying the first failure. It then identifies additional failures through subsequent repair and test. We can implement single-conclusion diagnosis in one of two ways. Either the inference system reports a conclusion as soon as it can infer one, or inference on premises and intermediate conclusions continues until it has gathered all information. In the latter case, we impose additional inference rules to limit the search space to yield a single conclusion. In *multiple-conclusion* diagnosis, on the other hand, the system tries to identify as many faults as possible.

The model permits both forms of diagnosis. Tests are done until the inference system knows all test outcomes either through evaluation or inference. The system then examines the set of conclusions to determine which conclusions it can actually draw.

Multiple-conclusion diagnosis results quite naturally when the inference system considers only direct inferences from the dependencies in the model. Single-conclusion diagnosis results when the system eliminates conclusions and tests that are unrelated to a discovered failure indication.

In most cases, we assume single-conclusion diagnosis because the search space is significantly smaller. The model does permit a limited level of multiple-conclusion diagnosis under the single-conclusion assumption by mapping multiple failures as single-conclusion elements. In another article in this series, we will describe the implications of the single-conclusion assumption and the methods used to relax it.

Table 1. *First-order dependencies for the network in Figure 1.*

Test Element	Dependencies
t_1	int_1, c_1
t_2	t_1, c_2
t_3	t_2, t_4, c_4
t_4	t_1, t_6, c_3, c_9
t_5	t_4, c_6
t_6	t_4, c_7
t_7	t_3, t_5, t_6, c_5, c_8

Representing logical constructs

Test-to-test dependencies provide the framework for information flow through the system with conclusions at the start of various chains (or at the end if we are considering logic diagrams). To generate the dependency graph in Figure 1, we begin with a functional representation of the system and construct a typical functional block diagram. Such a diagram is a directed graph in which nodes define functions and edges define functional flow. Functional tests provide information about whether the functions are behaving as specified. They also measure the system's health at a given point. Thus, any function that feeds a functional test will affect the outcome of that functional test. To represent this relationship, we put tests in the appropriate flow paths of the functional block diagrams.

To interpret the corresponding logic diagram (Figure 2) for this system, we need to reverse the logic such that conclusions are drawn when the corresponding components function properly. Thus, a test premise that is "true" corresponds to a passing test. By normal inference, this truth value propagates down the chain, asserting all downstream tests and conclusions as true (functional). Thus, in the dependency framework, the failure of a test depends on all downstream elements of that test in the logic diagram. In either formulation, tests may depend on other tests or on conclusions, but conclusions do not depend on anything. That is, they are terminal nodes. Thus, when a conclusion fails, information about that failure flows through the network to the tests on the

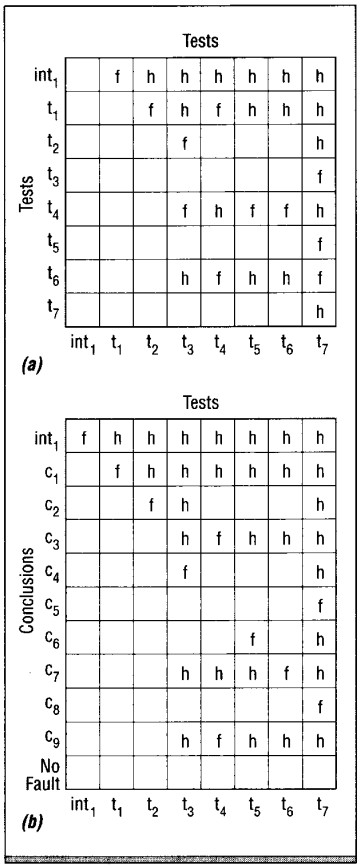

Figure 3. *Dependency matrices for the system represented in Figures 1 and 2. Test-to-test dependency matrix* ***(a)*** *and test-to-conclusion matrix* ***(b)****. f = a first-order dependency and h = a higher order dependency.*

dependency chain. Table 1 shows the dependencies for the network in Figures 1 and 2.

We can store this network using traditional data structures for representing graphs. We chose to represent test-to-test dependencies as a bit-adjacency matrix in which rows correspond to test feeds and columns to test dependencies. The test-to-conclusion dependencies are also represented as a bit-adjacency matrix. In this matrix, rows correspond to tests fed by conclusions, while columns correspond to tests that depend on conclusions.

Figure 3 shows the matrices corresponding to the sample system. Figure 3a shows the test-to-test dependency matrix, while Figure 3b shows the test-to-conclusion dependency matrix.

This form of representation is limited. Because of the way we have defined dependency, the matrix orientation forces a logical interpretation on the rows and columns. We can say that if a given conclusion is true (the corresponding component has failed), then all tests fed by that conclusion (the conclusion's row) are also true, unless we have asymmetries. This relationship is represented in the following logical form:

$$\text{conclusion}_i \supset \prod_j \text{test}_j \quad (1)$$

where test_j depends on conclusion_i, and $\prod$ represents conjunction. This form is also true if we know that a test has failed. In this case, all tests fed by the failed test must also fail.

$$\text{test}_i \supset \prod_j \text{test}_j \quad (2)$$

given test_j depends on test_i.

The columns in the matrices provide information about the possible cause of a test failure. Thus, if a test passes, then all elements in the corresponding column (both tests and conclusions) must also pass.

$$\neg \text{test}_i \supset \left(\prod_j \neg \text{test}_j\right) \wedge \left(\prod_k \neg \text{conclusion}_k\right) \quad (3)$$

given test_i depends on test_j and conclusion_k.

We may, however, want to have the corresponding logical expressions with the connectives reversed:

$$\text{conclusion}_i \supset \sum_j \text{test}_j \quad (4)$$

$$\text{test}_i \supset \sum_j \text{test}_j \quad (5)$$

$$\neg \text{test}_i \supset \left(\sum_j \neg \text{test}_j\right) \vee \left(\sum_k \neg \text{conclusion}_k\right) \quad (6)$$

where Σ represents disjunction. We do not address Equations 4 and 5 directly but provide methods for using the conditional test and for mapping special elements into the matrix.

Equation 6 provides for including multiple conclusions in the model. We create a special conclusion, called a failure group, and make a test depend on that group. The result is that the row corresponding to this conclusion contains all test feeds from the corresponding individual conclusions as well as the test that depends on the group. Thus, all members of the group must fail for the test to fail.

The following example illustrates how to include a multiple failure in the model. Assume three possible failures in a computer: a burnt-out disk-drive motor, a defective read/write head, and corrupt disk media. The burn-out may or may not have caused the read/write head to crash into the disk media. If we simply define a test that depends on the motor, the read/write head, and the disk media, then if one of these three fails, the test fails. On the other had, if we want a test to fail only when all three faults exist, we must specify a multiple failure group of the three faults. We can then define a test that depends on the group.

The result is the addition of a new primitive conclusion to the model. The new test depends only on the new conclusion rather than the members of the group, and all tests that depend on the elements of the group also depend on the group. Figure 4 illustrates this idea.

Weighting factors

Our goal in generating a diagnostic strategy is to minimize some set of resources and still effectively isolate the cause of the problem. Tests are often selected on the basis of how much information they provide, thus minimizing the number of tests required. Frequently, however, this approach is inadequate. Selected tests may be extremely difficult to perform or require a lot of time to complete. Thus, we need a way to select tests that considers available resources as well as the amount of information. Our approach is to weight the test-information yield according to several cost or probability criteria.

A *direct weight* is applied directly to the tests in the model. An increase in direct weight makes the test less desirable. An *indirect weight*, on the other hand, is attributed only to tests in the model

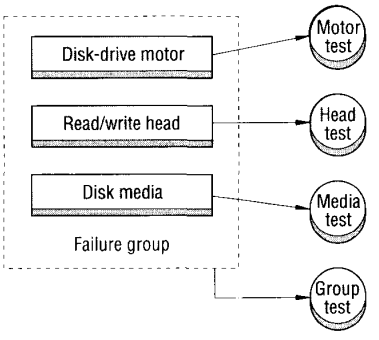

Figure 4. *Multiple-failure group with test dependencies.*

through a calculation based on the conclusions in the model. In this weighting, an increase in a value associated with a conclusion will make all tests that depend on that conclusion more desirable. We apply indirect weights directly to the conclusions and determine the corresponding test weight by examining the dependencies of the tests on the conclusions.

For each test in the model, we assign cost weights. A cost weight may indicate any test factor such as time to test, cost of resources, and skill-level requirements. In general, we select weights to be inversely proportional to the desirability of doing the test from a cost perspective. These weights define the direct weighting factor for the model.

We also assign weights to the conclusions in the model We select them in a way that makes them directly proportional to the desirability of identifying that conclusion. Some appropriate weights include how often a component fails and how critical a component is to mission completion. These weights define the indirect weighting factor for the model.

Related elements

In addition to the individual primitive elements—tests and conclusions—groups of tests and conclusions are important to our diagnostic model. A *test group* includes tests that have some logical relationship. For example, tests may require the same piece of equipment or may be accessed in the same physical location in the system. By grouping the tests together, we can generate a diagnostic strategy that remains within a group until we have all the required information. Once the test group has been tested, the normal process of selecting tests resumes.

Conclusions are grouped according to types. The first type is called a *replaceable unit group*, which specifies a higher level fault-isolation conclusion. Since systems are frequently subdivided, the replaceable unit group permits two levels of detail in a single model. For example, we can construct a model to the component level but include grouping at the SRU (shop-replaceable unit) level.

Alternatively, we can construct models in which primitive conclusions are failure modes and groups correspond to specific diagnoses that require system repair. For example, a resistor may have one of three failure modes. It may fail open, be shorted, or be out of tolerance.

The model allows test groups to be sequenced but provides for overrides to accommodate different types of test ordering.

We can combine these failure modes into a replaceable unit group that corresponds to a single resistor. If one of the three failures occurs, the failure will identify the resistor group, and we can replace the resistor.

Within the framework of an inference system, the replaceable unit group defines where test selection stops. Normally, the maintenance technician selects and evaluates tests until the inference system isolates a primitive element or ambiguity group. When the testability analyst groups the primitives as replaceable units, test selection continues until the system isolates a group. We no longer need to test down to the primitive level. In addition, we can conduct testing in stages in which the replaceable unit group is isolated first. Once the inference system identifies a group, it reports it. After isolating the group, the system continues selecting tests down to the primitive level. When the primitive conclusion is drawn, it too is reported.

The second type of conclusion group within our model is the *failure group*. The failure group is a group of conclusions that we expect to fail simultaneously. Because we assume that only a single conclusion will be drawn, we specify multiple conclusions as a single failure group. This permits a limited form of testability analysis for multiple failures and the isolation of faults from multiple failures. During analyses, we treat failure groups as primitive elements. We synthesize these new elements from the previous primitives that are members of the group, thus handling the failure group directly. In fact, if a test depends on a multiple failure group, the group is actually mapped as a primitive element directly into the model.

Test-sequencing declarations

One of the goals in developing the diagnostic model is to derive a diagnostic strategy that effectively and efficiently uses a system's testability. We want to be able to identify all failures as they occur, and we want to identify only the elements that have failed. In addition, we want the process used to test a system to be inexpensive in terms of any number of cost criteria.

Our model provides the basis for a method to develop such a strategy. Nevertheless, at times we need to override optimization to choose tests in an order that makes sense.

For example, suppose a technician is testing a card on an ATE test station. Assume that the ATE performs each test in approximately the same time and with the same reliability. Further assume that all tests are automatic. Under this scenario, we can use the model to select

tests to minimize the average number of tests evaluated.

A technician typically tests a unit under test, such as an electronic circuit, after running a set of tests to determine that it is safe to apply power to the UUT. These tests are called safe-to-turn-on tests. But these tests provide little diagnostic information. A selection approach based purely on theoretical information would not be likely to select these tests unless absolutely necessary. Common sense, however, dictates that these tests should be run first. Thus, the model needs a mechanism for overriding the information-based choice and choosing the group of safe-to-turn-on tests.

Our model allows test groups to be sequenced. If the technician selects the option to sequence test groups, then one or more test groups must be defined in the model. The test choice algorithm then selects the first test group, chooses all needed tests from this group, chooses the next test group, selects all needed tests, and continues in this fashion until it has exhausted all the test groups. It then considers any remaining (ungrouped) tests for evaluation.

Using this approach, the analyst defines a test group containing the safe-to-turn-on tests and then invokes the test-group-sequencing option. The algorithm chooses the safe-to-turn-on tests first and then the remaining tests in the model.

At times, there is a need to order a set of tests within a group. Tests for a sequential circuit, for example, may be sensitive to the current state of the circuit (values in the flip-flops). Many test programmers write tests to follow a sequence of states. If a technician has to test a portion of the circuit according to a sequence, then the analyst must order the corresponding tests to maintain the appropriate system state throughout the test process. This ordering is another optimization override that may be included in the model.

Of course, ordering operations are not limited to the sequencing options presented so far. Sheppard examined Allen's temporal calculus[13] and developed a mapping of 18 temporal relations into the modeling framework.[14] He also developed an algorithm that efficiently chains temporal relations. With this approach, modelers can specify first-order temporal relations at the same time they specify the first-order model.

Additional order operators, such as test overlapping and test inclusion, are also possible because the algorithm examines potential time intervals in which tests are performed. It also provides for extensions to include concurrency among test events as well as test prerequisites to drive test selection.

Analyzing the model

Several types of analyses are possible with an information-flow model, including determining testability characteristics, generating fault-isolation strategies, and evaluating strategies generated by other approaches. Figure 5 provides a flow chart describing the complete analysis process. The first step in analyzing the model is to develop a higher order representation through logical chaining.

Logical chaining

Once we have developed an information-flow model of a system, we preprocess it to determine higher order implications. Part 2 in this series of articles will describe this preprocessing in more detail.

Testability characteristics

After logical chaining, we are in a position to assess system testability. There are many ways to measure a system's testability characteristics, both directly and indirectly. The first step is to identify problem areas. Our model identifies (among others)

- ambiguous failures
- undetected failures
- information-flow feedback
- operational isolation

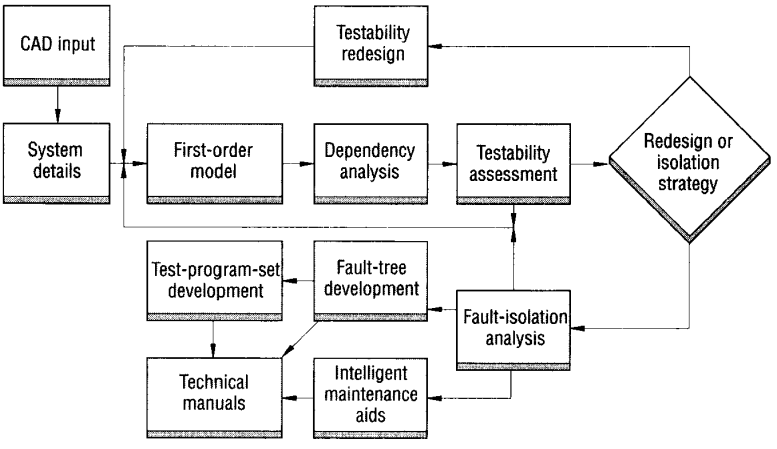

Figure 5. *Overview of testability and diagnostic analysis.*

- potential for hidden failures or false failures
- tolerance of false alarms
- use of test resources

Ambiguous failures. When the defined set of tests cannot distinguish between two or more failure conclusions, the failures are ambiguous. Our model is concerned with two levels of ambiguity. The first is ambiguity between primitive conclusions. For every conclusion, we define a *failure signature*. The failure signature is the set of tests that we expect to fail if the conclusion is true. If two or more conclusions have the same failure signature, they are ambiguous. If this condition exists, then regardless of the testing level, the defined set of tests will never distinguish between the two conclusions. This type of ambiguity is only a potential deficiency in the system, however, because our concern may be at a higher level.

The second level of ambiguity is ambiguity between two replaceable unit groups. Group ambiguity arises when two or more conclusions in the same ambiguity group are members of two or more replaceable unit groups. If testing isolates the ambiguous conclusions, we do not know which of the replaceable unit groups to repair or replace. But if an ambiguity group is wholly contained in one replaceable unit group, and we are interested only in isolating the fault to the replaceable unit group, then this ambiguity is benign.

Clearly, we want to identify ambiguity within the system. If analysis occurs early enough in development, we may be able to design additional tests to break up the ambiguity. If we are interested in testing only at the group level, we may be able to repackage the components of the system to contain any ambiguity within a replaceable unit. In this case, we do not need to define additional tests.

Undetected failures. We say a failure is not detected if none of the tests fail when the system has a fault. By examining the failure signatures of the conclusions, we can identify cases in which failures are not detected. If a conclusion has a failure signature that has no tests, then that conclusion is a case in which the failure is not detected. The model has a special conclusion called No Fault. The No Fault conclusion has no dependent tests, so the condition in which failures are not detected will be ambiguous with No Fault. Conditions in which failures are not detected frequently arise when the set of tests at one level (say the LRU level) detects a failure, and the unit containing the failure is pulled for further testing. Yet the tests defined at the next level (say the SRU level) do not detect the same failure. This leads to a RTOK (retest okay) situation, thus indicating a deficiency in the complete maintenance system.

We must improve testability to either trap the multiple failures or separate them so that no false failure occurs.

Information-flow feedback. When diagnostic information feeds back on itself to form a cycle, then all conclusions directly related to tests in the cycle will appear to be ambiguous. For example, suppose we have three tests t_1, t_2, and t_3 and three conclusions c_1, c_2, and c_3. Suppose t_1 depends on c_1 and t_3; t_2 depends on c_2 and t_1; and t_3 depends on c_3 and t_2. After prechaining, each test depends on the other two tests and all the conclusions. Thus, if c_1, c_2, or c_3 fails, the same failure signatures will result. So the three conclusions are ambiguous.

Generally, such circularities result from poorly defined tests or errors in the model. Sometimes the feedback loops required for proper system performance also lead to information-flow feedback. We must be careful when breaking these loops. We cannot permanently break required loops for functionality, and we must ensure that additional test hardware or procedures do not degrade the system performance.

Operational isolation. Operational isolation is the percentage of fault-isolation events that result in the ambiguity of n or fewer replaceable units. Frequently, design specifications include the requirement that fault isolation be to n or fewer replaceable units x percent of the time. We arrive at measures for operational isolation by considering the sizes of the ambiguity groups and the probabilities of isolating faults to particular ambiguity groups. These measures have a direct impact on logistics support planning for the system analyzed.

Potential for hidden and false failures. These two testability problems concern how to isolate a fault when multiple failures occur. The first problem is to analyze the potential for hidden failures—when the failure of one element masks the failure of other elements—which is a natural consequence in complex systems.

Masking can cause serious maintenance problems when one of the masked items is a *root-cause failure*, which results in a secondary failure of another system component. If we cannot isolate the root cause, we cannot effectively repair the system, since repairing a secondary failure is futile. The

secondary failure recurs when the system is reinitialized with the same symptoms as before. We are then in a repair circularity that will not return the system to operational status.

If we find no root-cause relationship, we can locate multiple failures and repair the system by repeat isolation. If we establish a root-cause relationship, we need to improve the testability. If we establish a tight root-cause relationship, such as when failure of the secondary element occurs if and only if a failure of the root cause happens, then we can handle the problem by annotating the repair procedures to include repair of both the root-cause and secondary failures. Without this tight relationship, we need tests to separate the root-cause from the secondary failure.

The second problem in analyzing multiple failures is the potential for false failures. A *false failure* occurs when the symptoms of two or more failures add up to mimic the failure of an unrelated element. Repairing the indicated failure has no effect, and the system is not put into an operational status. In other words, we have not identified the faulty components, and maintenance leaves the system in the original failed state. We must improve testability to either trap the multiple failures or separate them so that no false failure occurs.

Tolerance of false alarms. If we are concerned about the test set itself, we need to determine the tolerance of false alarms. Some tests may indicate a problem when no problem exists because of exceedingly tight test tolerances, inability to fully measure the required parameters, or inadequate understanding of the mechanisms and dependencies involved in taking test measurements. When field false alarms manifest themselves, two principal actions are taken: opening the test tolerances, which attacks the first problem but often at the expense of missed detections, and repeat testing, which is designed to allow transients to die down before the final evaluation. Repeat testing often takes the form of "n occurrences of anomalous behavior within y milliseconds for a bad test outcome."

Another action, often overlooked, is a common-sense check of the form "if test t_a indicates an anomalous reading then test t_b should reflect that problem also." This action requires some understanding of the system. The false-alarm tolerance, which we compute by examining the test-to-test dependencies, provides a measure of the system's ability to verify previous test outcomes. If this ability to verify previous test outcomes is sufficient, we may invoke consistency checking in the isolation strategies when false alarms are a problem. When it is insufficient we may wish to add extra tests to provide a margin for field false alarms.

Don't eliminate tests solely because they are excess or redundant. Instead, examine the total impact on maintenance.

Use of test resources. When analyzing a system in the design phase, we may still need to develop the tests through detailed test specifications. These test specifications are normally put together in a test requirements document, which is often a costly and time-consuming process. Before actually writing the elements of the TRD, we may want to ascertain which tests are contributing to fault isolation and which are not. Some tests under development may not contribute anything to fault isolation and can therefore be eliminated. Moreover, we may need additional tests because of ambiguity within the system, inadequate operational isolation, or frequently occurring false alarms.

We can identify some of these problems during design, but others, such as false-alarm tolerance, may require the system to be operational.

When considering tests for elimination, we need to identify two types of tests. An *excess test* provides the same diagnostic information as some combination of other tests. A *redundant test*, which is a special form of excess test, provides the same diagnostic information as some other single test. In information-flow feedback, we found that all conclusions in the feedback loop are ambiguous. Thus, all tests in the feedback loop are redundant because all of them examine the entire feedback loop.

We must be careful in eliminating redundant or excess tests from the model and thus from the diagnostic system. The elimination of excess tests will most likely reduce the tolerance to false alarms. If false alarms are a concern, excess tests may provide invaluable assistance in minimizing inappropriate action when a false alarm occurs; however, including excess tests may reach a point of diminishing returns in providing the necessary cross-checks.

Another concern in removing these tests is that each test may have a different cost. For example, test t_a and test t_b may be redundant, but t_a may be inexpensive to perform and somewhat unreliable while t_b is highly reliable but difficult to perform. We could choose either t_a or t_b, depending on the available resources and how critical each was to mission completion.

Finally, removing excess tests may increase the problems with isolating faults when there are multiple failures. The potential for false failures may increase be-

cause tests no longer exist to differentiate the symptoms. A good strategy is not to eliminate tests solely because they are excess or redundant, but to analyze the total impact on maintenance.

Developing diagnostic strategies

The purpose of any testability analysis is to provide the system with adequate diagnostics. After the analyses just described, defining a diagnostic strategy is the next logical step.

Mathematically, fault isolation is a set partition problem. Let $C = (c_1, c_2, ..., c_n)$ represent the set of components. After the jth test, a fault-isolation strategy partitions C into two classes:

$$\mathbf{F}^j = (c_1^j, c_2^j, ..., c_m^j)$$

which is the set of components that are still failure candidates after the jth test (feasible set), and

$$\mathbf{G}^j = \mathbf{C} - \mathbf{F}^j$$

which is the set of components found to be good after the jth test (infeasible set). By this structure, a strategy will have isolated the failure when $\mathbf{F}^j$ consists of a single element or a component ambiguity group that cannot be divided.

Test results impart information. Any diagnostic strategy must consider the type, amount, and quality of such information. In our model, we assume that what the test indicates (good or bad) actually reflects the state of the UUT. If we know all the dependencies in a system, we can calculate the information content of each test. When a test is performed, the set of dependencies allows us to draw conclusions about a subset of components.

At any point in a sequence of tests, we can compute the set of remaining failure candidates. The algorithm we have developed looks at the information content to minimize the average number of tests required to isolate a fault over the set of potential failure candidates. This adaptive approach embodies several artificial intelligence algorithms, including inference and pattern recognition.

Let $\mathbf{D}$ represent the full set of dependency relationships between components and test points as a matrix. $\mathbf{S}_k$ is a sequence of k tests, $(t_1, t_2, ..., t_k)$. $\mathbf{F}^k$ is the feasible failure candidate set associated with $\mathbf{S}_k$. An information measure $\mathbf{I}_j^k$ for each unperformed test t_j is a function of the dependency relationship and the remaining candidate failure class, say, $\mathbf{I}_j^k = f(\mathbf{D}, \mathbf{F}^k)$.

We obtain $\mathbf{S}_k$, which is based on an unknown current outcome, by optimizing at each decision point. That is, we take the next test in the sequence as the test that maximizes $\mathbf{I}_j^k$ for the conditions imposed by each previous test outcome. The sequence ends when we have enough information to isolate the fault.

Although this algorithm uses the greedy heuristic, it is based on a higher order representation and has been providing performance near the theoretical optimum. The algorithm also allows for weighting by individual resource factors.

Evaluating diagnostic strategies

An added benefit of our information-modeling approach is that we can evaluate existing diagnostic strategies in the form of fault trees. Rather than using the test-choice algorithm described earlier, we use the tests specified in the fault tree and compute the corresponding inferences with the model and the inference system. Using the inference system, we can directly compute the following information:

- actual isolations for each path in the tree including conflicts or resulting ambiguity groups
- average cost and expected cost for the strategy based on supplied data on test cost
- average time and expected time for the strategy based on supplied data on test time
- a detailed analysis of test cost, test time, and failure frequency for each branch in the tree

If we do not have a model and assume that the available strategy is correct, we can use paths in the fault tree to determine failure attributes of the system. These attributes can provide an initial model. However, we are still developing this analysis technique.

Interactive diagnosis

We can also develop fault-isolation strategies in a dynamic environment, as information is obtained during testing. This type of strategy, called adaptive testing, is sensitive to the context of the problem. Interactive maintenance aids, automatic test equipment, and embedded maintenance systems are tools for adaptive testing.

Interactive maintenance aids

Interactive maintenance aids are usually either electronic manuals, intelligent maintenance aids, or semiautomatic maintenance aids. Their capability is limited to how well we can electronically manipulate the diagnostic process using today's computing technology. Portable diagnostic aids may be strictly electronic manuals or intelligent maintenance assistants, while large computer systems could use large-scale simulation to guide diagnosis. Our focus is the environment of the small computer.

Technical information devices are machine representations of technical manuals in the form of electronic manuals that use static fault-isolation strategies.

These strategies display test procedures in proper order for fault isolation by following the fault tree and "turning the manual's pages" for the technician. When the fault is isolated, they present repair procedures. These devices have minimal machine requirements but provide little in terms of flexibility, training, or logistics support. They are also unable to learn from repeated applications.[15]

Intelligent maintenance aids are computer programs with extensive knowledge of the UUT. These devices have the potential for considerable flexibility because they can compute the next maintenance action using the known information and the problem context. Obtaining the desired level of flexibility, however, takes some effort unless we have done compatible testability modeling of the system. Intelligent maintenance devices tie a model to an electronic manual's database to form an interactive, context-sensitive maintenance aid. The model-based approaches described here are useful in exploiting the flexibility of these devices.[16]

Semiautomatic maintenance aids are intelligent maintenance assistants that can query other databases for answers to test questions (such as embedded BIT) or can execute and interpret tests. They are connected to the system being analyzed for downloading information or measuring through stimulus-response testing. Fault isolation in semiautomatic maintenance aids is similar to that in intelligent maintenance aids except that semiautomatic aids can answer many of their own questions.

Automatic test equipment

With only a small extension, the semiautomatic maintenance aid can become intelligent ATE. ATE systems are developed for second- and third-level maintenance and typically consist of an instrument suite, an interface test adapter, a computer, a test executive, and a test program set. The TPS, which defines how to diagnose a piece of equipment, is usually developed using static, predefined fault trees. But because the static fault tree has limited flexibility, we generally require a limited amount of intelligence.

Intelligent ATE considers known information such as BIT readings, operator complaints, and logistical history. It also adapts to changing conditions during maintenance and provides sufficient flexibility for the technician to work around deficiencies in test equipment or other factors. An example of intelligent ATE that Arinc has developed is a system that uses a Racal-Dana VXI automatic test unit to diagnose faults in power supplies on the AV-8B (Harrier) aircraft.[17]

Embedded diagnostics

The most significant application of the model-based approach has yet to be realized. Many complex system architectures have evolved to the point of having on-board processors, on-board measurement devices, bulk memory, and data buses for information transfer. One such architecture for avionics systems includes all of these elements and some reconfiguration capability.[18] These hardware developments, together with the modeling approaches described in this article, make possible an embedded diagnostic system that can examine its own maintainability and testability. The system then diagnoses possible failures and reconfigures itself to complete mission requirements.[19]

We have covered some of the basics in information-theoretic approaches to integrated diagnostics. We have described how to create a mathematical structure that is both hierarchical and technology independent. This structure, however, is not a global solution to diag-

Table 2. Results of applying information-flow modeling to integrated diagnostics.

System	Customer	Results
Air pressurization system	International Fuel Cell	Improved unique isolation by 100%
AN/MSQ-103C Teampack	EW/RSTA/USA	Reduced required testing by 87% and developed a portable maintenance aid
Mk 84 60/400 Hz Static	Navsea/USN	Reduced required testing by 70% and developed a portable maintenance aid
UH-60A Stability Augmentation System	ATL/USA	Reduced mean time to isolate fault by 90%; reduced maintenance complexity by 70%
ALQ-131 Podded EW System	ASD/USAF	Reduced mean time to isolate fault by 75%
ALQ-184 Podded EW System	AFLC/USAF	Reduced false-alarm rate by 90%; developed software test procedures for unit under test
B-2 Bomber DFT Program	USAF/Northrop	Improved specification compliance at the shop-replaceable-unit level by 80%

nosis because it discards much of the detail about a particular test's worth—detail that is needed to develop a reliable and complete diagnostic system.

Our modeling approach uses information as the medium of exchange and information fusion as the inference process. It handles imprecise testing by reasoning under uncertainty and has been used for the analysis of testability and fault diagnosis, diagnostic strategies, and maintenance aids.

We have applied this approach to more than 200 systems and have repeatedly demonstrated the effectiveness of information modeling in improving field performance. Table 2 lists a few of our successes. The results in the table are consistent with the results we have achieved overall. Improvements have been significant in most cases and occasionally spectacular. D&T

References

1. M. Garey, "Optimal Binary Identification Procedures," *J. Applied Mathematics*, Vol. 23, No. 2, Sept. 1972, pp. 173-186.
2. L. Hyafil and R. Rivest, "Constructing Optimal Binary Decision Trees in NP-Complete," *Information Processing Letters*, Vol. 5, No. 1, May 1976, pp. 15-17.
3. B. Moret, "Decision Trees and Diagrams," *Computing Surveys*, Vol. 14, No. 4, Dec. 1982, pp. 593-623.
4. K. Pattipati and M. Alexandridis, "Applications of Heuristic Search and Information Theory in Sequential Fault Diagnosis," *IEEE Trans. Systems, Man, and Cybernetics*, July/Aug. 1990, pp. 872-887.
5. G. Cross, "Third Generation MATE—Today's Solution," *Proc. Automatic Test Conf.*, IEEE Press, New York, 1987, pp. 289-292.
6. C. Espisito et al., "US Army/IFTE Technical and Management Overview," *Proc. IEEE Automatic Test Conf.*, IEEE Press, New York, 1986, pp. 319-322.
7. M. Najaran, "CASS Revisited—A Case for Supportability," *Proc. IEEE Automatic Test Conf.*, IEEE Press, New York, 1986, pp. 323-327.
8. W. Keiner, "A Navy Approach to Integrated Diagnostics," *Proc. IEEE Automatic Test Conf.*, IEEE Press, New York, 1990, pp. 443-450.
9. *Testability Program for Electronic Systems and Equipment, MIL-STD-2165*, Naval Electronic Systems Command (ELEX-8111), Washington, DC, 1985.
10. J. Sheppard and W. Simpson, "Integrated Maintenance—A Hierarchical Approach," *Proc. IEEE Automatic Test Conf.*, IEEE Press, New York, 1990, pp. 477-483.
11. A. Mosely, "Why an Air Force Centralized Integrated Diagnostics Office?" *Proc. IEEE Automatic Test Conf.*, IEEE Press, New York, 1990, pp. 373-376.
12. W. Simpson and B. Kelley, "Mathematical Formulation of Fault Isolation," STAMP tech. note 330.1, Arinc Research Corp., Annapolis, Md., 1986.
13. J. Allen, "Maintaining Knowledge About Temporal Intervals," *Comm. ACM*, Vol. 26, No. 11, Nov. 1983. pp. 832-843.
14. J. Sheppard, "Applying Propositional Calculus to Temporal Reasoning," AI tech. note 1301, Arinc Research Corp., Annapolis, Md., 1989.
15. W. Simpson, "Active Testability Analysis and Interactive Fault Isolation Using STAMP," *Proc. IEEE Automatic Test Conf.*, IEEE Press, New York, 1987, pp. 105-111.
16. J. Sheppard and W. Simpson, "Incorporating Model-Based Reasoning in Interactive Maintenance Aids," *Proc. Nat'l Aerospace Electronics Conf.*, IEEE Press, New York, 1990, pp. 1238-1242.
17. W. Simpson and J. Sheppard, "Developing Intelligent Automatic Test Equipment," *Proc. Nat'l Aerospace Electronics Conf.*, IEEE Press, New York, 1991, pp. 1206-1213.
18. *JIAWG Advanced Avionics Architectures J87-01 (CAB II)*, Joint Integrated Avionics Working Group, Dayton, Ohio, 1989.
19. E. Esker, W. Simpson, and J. Sheppard, "An Embedded Maintenance Subsystem," *Proc. IEEE Automatic Test Conf.*, IEEE Press, New York, 1990, pp. 331-336.

Direct questions or comments on this article to either author at Arinc Research Corp., Advanced Research and Development Group, 2551 Riva Rd., Annapolis, MD 21401; Sheppard's e-mail address is sheppard@cs.jhu.edu.

William R. Simpson is a research fellow in the Advanced Research and Development Group at Arinc Research Corp., where he is involved in testability and fault diagnosis. He helped develop the System Testability and Maintenance Program, which is based on an information-flow model. He was also a principal developer of Pointer, an intelligent, interactive maintenance aid. He holds a BS from Virginia Polytechnic Institute and State University and an MS and a PhD in aerospace engineering from Ohio State University.

John W. Sheppard is a senior research analyst in the Advanced Research and Development Group at Arinc Research Corp.. He is also pursuing a PhD in computer science at Johns Hopkins University. His research areas include applying AI techniques to fault diagnosis, machine learning, neural networks, and nonstandard logic. He has developed algorithms to diagnose system failures, verify knowledge bases, and classify software. He was also a principal developer of Pointer and assisted in the development of a prototype expert system that diagnoses system failures and reconfigures the system to keep functioning. He holds a BS from Southern Methodist University and an MS from The Johns Hopkins University—both in computer science.