Performing Effective Fault Isolation in Integrated Diagnostics

JOHN W. SHEPPARD
WILLIAM R. SIMPSON
Arinc Research Corporation

In the sixth and final article of their series, the authors develop fault trees, using an entropy-directed search process without modifications or constraints. Then they add factors typically encountered at different levels of fault isolation to modify and constrain the search process. Finally, they develop new fault trees to illustrate the impact of these factors.

IN TODAY'S CLIMATE of decreasing budgets, the number of new systems being developed is decreasing, and the lives of existing systems are being extended. Consequently, aging systems are experiencing failures more frequently than before. New systems also fail, and those failures are hard to diagnose because of the increased sophistication and complexity of the systems. Performing fault isolation on existing systems and new complex systems requires meticulous bookkeeping and detailed understanding of failure behaviors. The test engineer must devise approaches to fault isolation that provide accurate diagnostics and that take into account such factors as sequence, groups, and potential false positives and false negatives. In part 6 of our series on integrated diagnostics,[1-5] we construct several fault trees that account for these and other factors.

We present the case study of an antitank missile launcher, used throughout the series, in the context of a complete maintenance architecture. Case study documentation is not repeated here due to space limitations. The dependency diagram and the closed dependency matrices for the case study system appear on pages 53 and 54 of part 5 of the series.[5] Tables 1 and 2 on pages 26 and 27 of part 2[2] provide pertinent data for tests and conclusions. We use these data to develop diagnostic strategies.

Information flow model

This series has presented a model-based approach to integrated diagnostics. The information flow model permits a thorough design-for-testability procedure and a basis for generating efficient and effective fault isolation strategies. The information flow model defines the interrelationships of system elements in terms of the flow of diagnostic information. The model can include as an information source any event or observation that provides information about the system under study (for example, stimulus-response pairs, boundary-scan outputs, and probe information). The model can include as a fault isolation conclusion any conclusion that one can draw during diagnosis (a failure of a specific component, a specific failure mode of a piece of hardware, a nonhardware failure such as bus timing, and the absence of a failure). The model-based approach is hierarchical, and any single model can include any conclusion type.

The first step in the model-building process is to determine the level of diagnostic analysis required. Analysis could include embedded diagnosis (as in built-in test); manual organic maintenance; manual, semiautomatic, or automatic shop repair of units from field sites; or depot repair of expensive cards for resale or insertion into a logistics pipeline. The level of diagnosis determines the fault isolation conclusions to consider and the appropriate tests to conduct. Ideally, the engineer develops a hierarchy of sub-

0740-7475/93/0600-0078$03.00 © 1993 IEEE

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}	t_{18}	int_1	int_2	$t_i \in$ UI	Pass	Fail	Min (Pass, Fail)	Rank*
t_1		0	0	1	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	Yes	1+4 = (5)	1+4+5–0 =10	5	10
t_2	1		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	Yes	1+1 = (2)	1+11+1–0 =13	2	5
t_3	1	0		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	Yes	1+3 = (4)	1+10+0–0 =11	4	9
t_4	0	0	0		0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	Yes	1+6 = (7)	1+3+4–0 = 8	7	19
t_5	0	0	0	1		1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	Yes	1+4 = (5)	1+8+1–1 = 9	5	11
t_6	0	0	0	0	0		1	1	1	1	1	1	1	1	1	1	1	1	0	0	Yes	1+5 = (6)	1+6+2–1 = 8	6	13
t_7	0	0	0	0	0	1		1	1	1	1	1	1	1	1	1	1	1	0	0	No (t_6)	Take values from t_6		6	14
t_8	0	0	0	0	0	1	1		1	1	1	1	1	1	1	1	1	1	0	0	No (t_6)	Take values from t_6		6	15
t_9	0	0	0	0	0	1	1	1		1	1	1	1	1	1	1	1	1	0	0	No (t_6)	Take values from t_6		6	16
t_{10}	0	0	0	0	0	0	0	0	0		1	1	1	1	1	1	1	0	0	0	Yes	1+6 = (7)	1+5+2–1 = (7)	7	20
t_{11}	0	0	0	0	0	0	0	0	0	0		1	1	1	1	1	1	0	0	0	Yes	1+7 = 8	1+4+2–1 = (6)	6	17
t_{12}	0	0	0	0	0	0	0	0	0	0	0		1	1	1	1	1	0	0	0	Yes	1+8 = 9	1+3+2–1 = (5)	5	12
t_{13}	0	0	0	0	0	0	0	0	0	0	0	0		1	1	1	1	0	0	0	Yes	1+11 = 12	1+2+0–0 = (3)	3	8
t_{14}	0	0	0	0	0	0	0	0	0	0	0	0	0		1	1	1	0	0	0	Yes	1+12 = 13	1+1+0–0 = (2)	2	7
t_{15}	0	0	0	0	0	0	0	0	0	0	0	0	0	0		1	0	0	0	0	Yes	1+13 = 14	1+0+0–0 = (1)	1	2
t_{16}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		0	0	0	0	No (t_{15})	Take values from t_{15}		1	1
t_{17}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1		0	0	0	No (t_{14})	Take values from t_{14}		2	6
t_{18}	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1		0	0	No (t_6)	Take values from t_6		6	18
int_1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1		0	Yes	1+0 = (1)	1+12+1–0 = 14	1	3
int_2	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0		Yes	1+0 = (1)	1+11+2–0 = 14	1	4

* May have other rankings depending on where ties are ranked

$$\text{Pass} = 1 = \sum_{i=1}^{[UI]} (\text{column value} = 1 \wedge t_i \in \textbf{UI})$$ Nonunique elements and diagonal elements shaded to indicate no counting

$$\text{Fail} = 1 = \sum_{j=1}^{[UI]} (\text{row value} + 1 \wedge t_j \in \textbf{UI}) + \sum_{k=1}^{[UI]} (\text{column and row value} \neq 1 \wedge t_k \in \textbf{UI}) - \sum_{i=1}^{[UI]} (\text{elimination creates ambiguity})$$

Figure 1. *Information counts for not-needed ranking in case study test-to-test matrix. Circled values are minimum information counts. (See part 5 for mathematics.)*

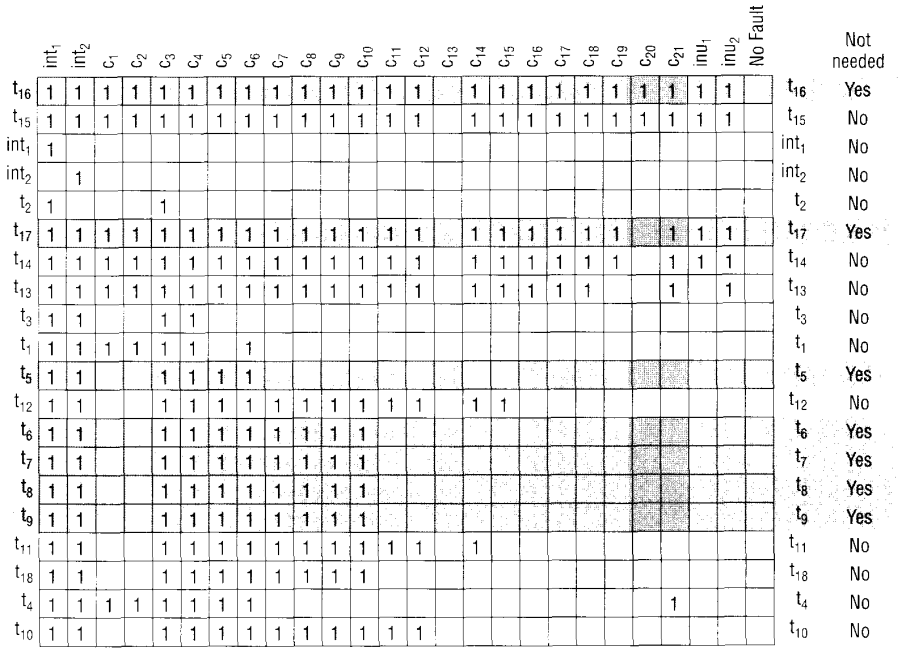
systems (thus, a hierarchy of models) to address each level of diagnosis. If the hierarchy is developed early in the system design process, the engineer can analyze the hierarchy and continue to refine system testability as the design matures. This approach fits well within concurrent engineering guidelines because of the close ties between system design and testability design.

Optimized fault isolation: the first step

Previously, we presented an entropy-directed search process and several inference rules for developing fault trees.[5] We derived entropy-directed search from Shannon's information theory,[6] which has also been applied extensively in decision theory and machine learning.[7,8] Entropy-directed search seeks to maximize the reduction of uncertainty independently of a test's outcome.[6,9] Using entropy-directed search, we reduce the search process to a count of information gained and focus only on learned test outcomes (except in the case of tiebreakers). The search for a conclusion terminates either when the outcomes of all tests are known or when only one viable conclusion remains.

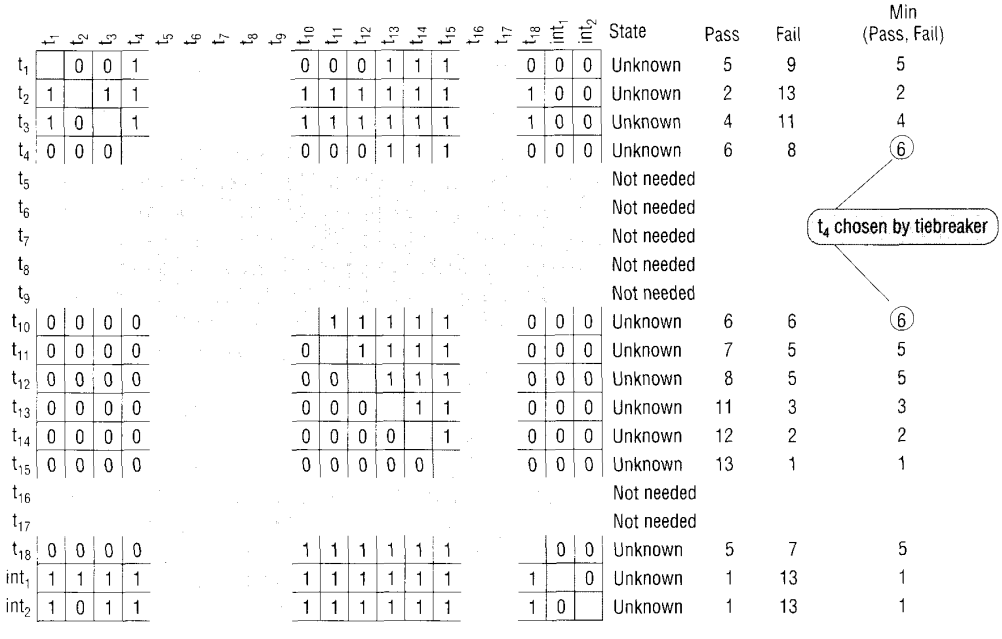
To construct a fault tree using entropy-directed search, we first reduce the computation load by determining which tests are not needed for diagnosis. The algorithm for determining which tests to eliminate is identical to the excess-test analysis algorithm presented in part 3.[3] We declare a test not needed if elimination of the test from the test set will cause no relevant additional ambiguity. (Ambiguity occurs when two conclusions have identical test signatures.) The algorithm ranks the tests in reverse order of desirability (based on information gain) and evaluates the least desirable test first. Because tests are evaluated sequentially and declaring a test not needed may affect whether downstream tests can be declared not needed, the ranking forces the least desirable tests to be eliminated first. This "preconditioning" of the test set gives us a tree with the minimum number of tests but may cause overall losses in efficiency.

Figure 1 shows the result of counting information gain for each test in the case



	int1	int2	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12	c13	c14	c15	c16	c17	c18	c19	c20	c21	inu1	inu2	No Fault		Not needed
t16	1	1	1	1	1	1	1	1	1	1	1	1	1	1		1	1	1	1	1	1	1	1	1	1		t16	Yes
t15	1	1	1	1	1	1	1	1	1	1	1	1	1	1		1	1	1	1	1	1	1	1	1	1		t15	No
int1	1																										int1	No
int2		1																									int2	No
t2	1				1																						t2	No
t17	1	1	1	1	1	1	1	1	1	1	1	1	1	1		1	1	1	1	1	1		1	1	1		t17	Yes
t14	1	1	1	1	1	1	1	1	1	1	1	1	1	1		1	1	1	1	1	1		1	1	1		t14	No
t13	1	1	1	1	1	1	1	1	1	1	1	1	1	1		1	1	1	1	1			1		1		t13	No
t3	1	1			1	1																					t3	No
t1	1	1	1	1	1	1		1																			t1	No
t5	1	1			1	1	1	1																			t5	Yes
t12	1	1			1	1	1	1	1	1	1	1	1	1		1	1										t12	No
t6	1	1			1	1	1	1	1	1	1	1															t6	Yes
t7	1	1			1	1	1	1	1	1	1	1															t7	Yes
t8	1	1			1	1	1	1	1	1	1	1															t8	Yes
t9	1	1			1	1	1	1	1	1	1	1															t9	Yes
t11	1	1			1	1	1	1	1	1	1	1	1	1		1											t11	No
t18	1	1			1	1	1	1	1	1	1	1															t18	No
t4	1	1	1	1	1	1	1	1															1				t4	No
t10	1	1			1	1	1	1	1	1	1	1	1	1													t10	No

Figure 2. *Not-needed analysis for case study.*



	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	t13	t14	t15	t16	t17	t18	int1	int2	State	Pass	Fail	Min (Pass, Fail)
t1		0	0	1						0	0	0	1	1	1			0	0	0	Unknown	5	9	5
t2	1		1	1						1	1	1	1	1	1			1	0	0	Unknown	2	13	2
t3	1	0		1						1	1	1	1	1	1			1	0	0	Unknown	4	11	4
t4	0	0	0							0	0	0	1	1	1			0	0	0	Unknown	6	8	(6)
t5																					Not needed			
t6																					Not needed			
t7																					Not needed			
t8																					Not needed			
t9																					Not needed			
t10	0	0	0	0							1	1	1	1	1			0	0	0	Unknown	6	6	(6)
t11	0	0	0	0						0		1	1	1	1			0	0	0	Unknown	7	5	5
t12	0	0	0	0						0	0		1	1	1			0	0	0	Unknown	8	5	5
t13	0	0	0	0						0	0	0		1	1			0	0	0	Unknown	11	3	3
t14	0	0	0	0						0	0	0	0		1			0	0	0	Unknown	12	2	2
t15	0	0	0	0						0	0	0	0	0				0	0	0	Unknown	13	1	1
t16																					Not needed			
t17																					Not needed			
t18	0	0	0	0						1	1	1	1	1	1				0	0	Unknown	5	7	5
int1	1	1	1	1						1	1	1	1	1	1			1		0	Unknown	1	13	1
int2	1	0	1	1						1	1	1	1	1	1			1	0		Unknown	1	13	1

Figure 3. *Information counts for first test choice in case study test-to-test matrix.*

study, together with the minimum information yield under an arbitrary outcome. We can weight the information gain for test times, skill levels, failure frequencies, or other factors before ranking. Figure 2 shows the result of evaluating the tests in ranked order. In the figure, the test-to-conclusion dependency matrix has been transposed and the tests reordered to reflect the ranking. The shaded rows indicate the tests to be declared not needed. For example, the first ranked test is t_{16}. If we remove that row from Figure 2 and compare the column signatures across all conclusions, we find that no two conclusions that previously had different signatures now have identical signatures. Thus, t_{16} is not needed and is eliminated. At this point, the algorithm masks t_{16} out of the matrix and evaluates t_{15}. Eliminating t_{15} would cause c_{20} to be identical to *No Fault* and c_{13}, so t_{15} is considered not excess and must be retained. Note that this is true only because t_{16} was eliminated. We see from Figure 2 that the analysis determines seven tests are not needed.

Once we have eliminated the excess tests (by assigning their test outcomes as not needed), we create a fault tree, using entropy-directed search with the depth-first tree traversal described in part 5. Figure 3 shows the test-to-test matrix after all known tests are masked and the information measure is computed for all remaining tests. The values in Figure 3 differ slightly from those in Figure 1 because different tests have been masked out. In Figure 1, six tests are masked out for uniqueness. In Figure 3, seven are masked out as not needed. The minimum information measure has a maximum value for both t_4 and t_{10}. We resolve the tie in favor of t_4 because it has the largest maximum information value. We assign this first test a pass outcome and provide both the test and its outcome to the inference engine (Figure 4). Because the termination metarules are not invoked, the search continues with a second test.

Figure 5 shows termination of the first path in the fault tree. At this point, only t_{15} remains, so we do not actually have to compute information gain. We must choose t_{15}. Drawing inferences from t_{15}'s passing invokes the termination metarules and assigns the diagnostic outcome to this path. Because the path has terminated, the algorithm backs up in the tree to t_{15}, which is assigned the next outcome (fail outcome). This also terminates the sequence. Again, the algorithm assigns the diagnostic outcome and backs up in the tree to t_{15}, which does not have an unevaluated outcome. The algorithm backs up again to t_{14}, which is assigned a fail outcome.

At this point, the sequence does not terminate, and the algorithm must choose a new test. Eventually, because there is no previous step, backing up fails, signaling completion of the tree. Table 1 shows the complete fault tree in tabular

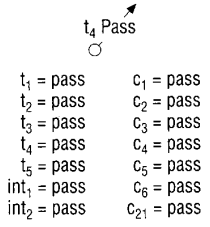


Figure 4. *First test of fault tree.*

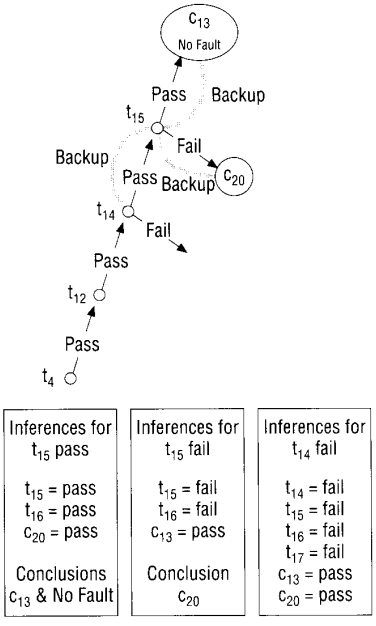


Figure 5. *Termination of a diagnostic sequence.*

Table 1. *Basic fault tree table for the case study.*

Step	Test	Previous step	Pass outcome	Fail outcome
1	t_4	0	Step 2	Step 9
2	t_{12}	1	Step 3	Step 6
3	t_{14}	2	Step 4	Step 5
4	t_{15}	3	c_{13}, *No Fault*	c_{20}
5	t_{13}	3	c_{19}, inu_1	c_{16}, c_{17}, c_{18}, inu_2
6	t_{10}	2	Step 7	Step 8
7	t_{11}	6	c_{15}	c_{14}
8	t_{18}	6	c_{11}, c_{12}	c_7, c_8, c_9, c_{10}
9	t_3	1	Step 10	Step13
10	t_{18}	9	Step 11	Step 12
11	t_1	10	c_{21}	c_1, c_2
12	t_1	10	c_5	c_6
13	t_2	9	Step 14	Step 15
14	int_2	13	c_4	int_2
15	int_1	13	c_3	int_1

Table 2. Skill-level-optimized fault tree for the case study.

Step	Test (skill level)	Previous step	Pass outcome	Fail outcome
1	t_{12} (E2)	0	Step 2	Step 7
2	t_{13} (E3)	1	Step 3	Step 5
3	t_{15} (E2)	2	c_{13}, *No Fault*	Step 4
4	t_{17} (E3)	3	c_{20}	c_{19}, inu_1
5	t_1 (E3)	2	Step 6	c_1, c_2
6	t_4 (E4)	5	c_{16}, c_{17}, c_{18}, inu_1	c_{21}
7	t_1 (E3)	1	Step 8	Step 12
8	t_{18} (E3)	7	Step 9	Step 11
9	t_{10} (E6)	8	Step 10	c_{11}, c_{12}
10	t_{11} (E7)	9	c_{15}	c_{14}
11	t_4 (E4)	8	c_7, c_8, c_9, c_{10}	c_5
12	t_2 (E3)	7	Step 13	Step 15
13	int_2 (E3)	12	Step 14	int_2
14	t_3 (E5)	13	c_6	c_4
15	int_2 (E3)	12	c_3	int_1

form. This tree is uniform in depth, always requiring four tests to isolate. We can use it to fault-isolate the case study, but we may need to include several practical factors, reviewed in the following sections.

Diagnosis in an operational environment

If we assume that the case study system is to be used in a field operational environment, we must verify that it is ready to perform or is repairable in this environment. We must consider two aspects of fault isolation in an operational environment. First, technicians of various skill levels attempt to isolate faults. It is therefore important to minimize the skill required. Second, we must consider the repair hierarchy. When a system problem is difficult to diagnose, the operational personnel send the system either to an intermediate repair facility or to a repair shop. For activities with remote operations, repair takes place at the replaceable unit level and not at the component level. Remote activities engaged in operations do not undertake repair but send the entire system to an intermediate repair facility. When diagnosis becomes too difficult or too many systems are sent back to the intermediate repair facility, it sends the system to a shop outside the operational environment.

Because of this repair hierarchy, we must develop three sets of fault isolation procedures and several different fault trees for the operational environment. The first set of procedures, for the intermediate repair facility and activities that are not remotely deployed, isolates to a level consistent with component repair. The procedures are based on minimizing either skill level or time to isolate, depending on personnel availability. The second set of procedures, to be used by remote activities, isolates to the replaceable unit level and is optimized for minimum skill level. The third set of procedures, for remote activities engaged in operations, determines whether or not the system is ready for use.

Weighted fault trees for intermediate repair. For intermediate repair of the case study, two isolation factors are important: skill level (designated by US military enlisted rank) and time.

In the field, an E7 may not be available to assist with each fault isolation, so a diagnostic procedure should use personnel of lower skill levels and call for an E7 only when needed. The first step in building a fault tree optimized for skill level is to assign skill-level weights to each test. For the case study, we assign the numeric values from the military skill-level designations, thus assuming a linear relationship between rank and skill level required for each test. If we believe the linear assumption is incorrect or if it yields unsatisfactory results, we can mathematically emphasize the skill-level factor. We detailed the analytic process of integrating multiple optimization criteria into the diagnosis, including the use of emphasis factors, in part 5.

The not-needed analysis applies appropriate weights (in this case, skill-level) before tests are ranked. Table 2 provides the tree that results from weighting by skill level (listed after each test name). Note that the system can be verified to be operational by an E3 (that is, all tests on the *No Fault* path require skill levels of E3 or less). Note also that only steps 9 and 10 require high skill levels. Further, all the difficult tests occur at the end of diagnostic sequences. This tree requires completion of an average of 4.31 tests—a slight increase over the unweighted tree, which requires 4.0 tests to isolate.

A second fault tree is needed for operational use when skill level is not a problem (that is, when E7 personnel are available) but time is. In some cases, however, the time required to perform a test is a function of a number of factors. For example, performing several tests in one sequence may require less time than performing the same tests in a different sequence. In that case, test time depends on previous tests performed. For time-critical applications, we may want to construct a matrix of test times as a function of previous tests. (Troy de-

scribes the matrix approach to time, cost, and other factors.[10]) In general, however, we define a time function in which test times depend on access panels that are already open, tools that are within reach, test equipment that has been set up, and so on.

For this article, we assume that a simple time metric is sufficient. For the case study, we wish to construct a fault tree for the intermediate repair facility to which personnel resources are fully available. Therefore, we construct a tree weighted for both test times and component failure frequencies so that the optimization goal is the minimum time to isolate the most frequently failing element. Table 3 shows the tree, with the test time for each test in parentheses. The expected time to fault-isolate is computed[5] as 3.11 time units; actual isolations vary between one and eight time units and require between three and six tests.

Table 3. *Time-optimized fault tree for the case study.*

Step	Test (time)	Previous step	Pass outcome	Fail outcome
1	t_{11} (0.10)	0	Step 2	Step 8
2	t_{13} (1.20)	1	Step 3	Step 5
3	t_{15} (1.50)	2	c_{13}, *No Fault*	Step 4
4	t_{17} (1.50)	3	c_{20}	c_{19}, inu_1
5	t_4 (1.50)	2	Step 6	Step 7
6	t_{12} (0.90)	5	c_{16}, c_{17}, c_{18}, inu_2	c_{15}
7	t_1 (2.00)	5	c_{21}	c_1, c_2
8	t_{18} (0.30)	1	Step 9	Step 10
9	t_{10} (0.60)	8	c_{14}	c_{11}, c_{12}
10	t_1 (2.00)	8	Step 11	Step 12
11	t_4 (1.50)	10	c_7, c_8, c_9, c_{10}	c_5
12	t_2 (2.20)	10	Step 13	Step 15
13	t_3 (2.40)	12	c_6	Step 14
14	int_2 (1.00)	13	c_4	int_2
15	int_1 (1.00)	12	c_3	int_1

Protecting intermediate repair from improper diagnosis. Some skill levels may not be available in some activities. As a result, personnel with lower-than-desirable skill levels may perform tests. This situation can lead to false test indications (data misinterpreted or tests incorrectly performed). If we can determine when the fault isolation may be in error, we can default maintenance to the next level of repair. So far, we have assumed that tests are completely reliable. For example, when we determine a test passes, the test really does pass. For a variety of reasons, however, the test may not be reliable. For example, to reduce the amount of code needed to support built-in test (BIT), we may have simplified the process for determining test outcomes, or we may have used less-sophisticated built-in test equipment (BITE) to reduce cost or weight. In manual diagnosis, the source of unreliable test outcomes may be as simple as not having personnel with appropriate skill levels. At the lower skill levels, we are not certain that test outcomes are being interpreted correctly or that the test is even performed correctly.

The false test indication discussed in the literature most frequently is the *false alarm*. (BIT false alarms and their implications for maintenance are discussed in reports by Malcom and by Sperry Corporation.[11,12]) The primary result of false alarms is wasted maintenance actions, including inappropriate repairs and inappropriate system downtime. If maintenance personnel repair the wrong unit, troubleshooting time will increase until the faulty unit is finally identified.

A second type of false test indication, rarely discussed in the literature but perhaps just as devastating, is the *false assurance*. In a false alarm, we have an indication of a problem through improper testing or test definition, but no problem actually exists in the system. In a false assurance, we have no indication of a problem although a problem actually exists. We can address a false assurance by determining whether certain tests are failing to provide expected information within the diagnostic process. Several solutions exist for improving individual tests, including repeat polling and modified tolerances.[13]

If the problem lies with the technician performing the tests rather than the tests themselves, we may wish to focus on verifying the conclusions being drawn. We can verify such conclusions by modifying the search process to choose tests that focus on the drawn conclusion as a hypothesis. Hypothesis-directed search and entropy-directed search differ in one important aspect: entropy-directed search does not presuppose any specific conclusion, but hypothesis-directed search does.

To perform hypothesis-directed search, we examine the test-to-conclusion matrix as shown in Figure 6 (next page). We now define two measures as follows:

$$e_i = \sum_{j=1}^{|F|} \alpha_{ij},$$

$$\alpha_{ji} = \begin{cases} 1; \left(t_i \text{ depends on } c_j\right) \wedge \left(c_j \in \mathbf{UF}\right), \\ 0; \text{ otherwise} \end{cases}$$

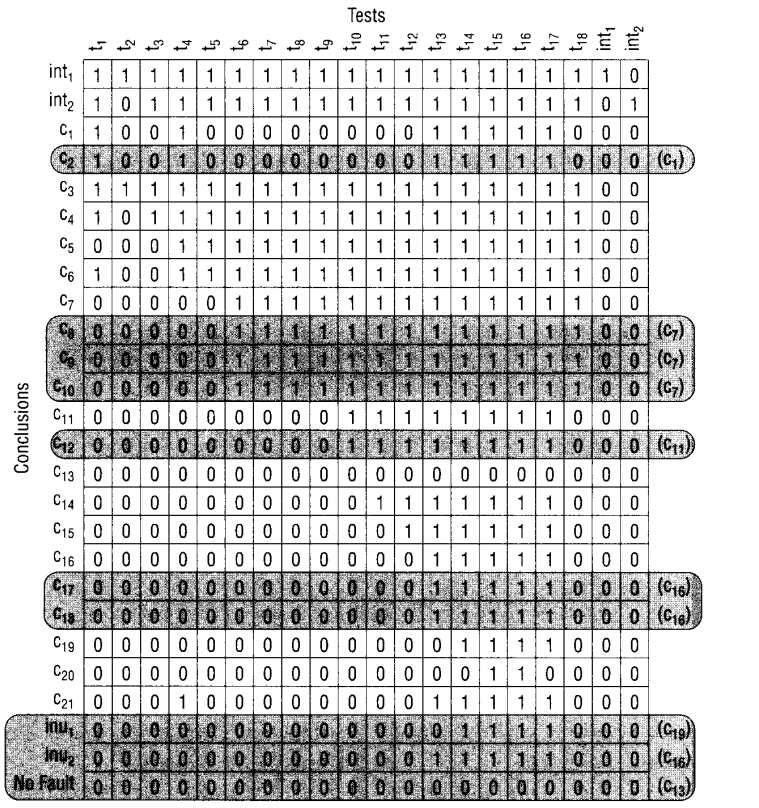


Figure 6. *Hypothesis-directed search data for case study.*

where e_i is the number of conclusion dependencies in t_i, t_i is the ith test (information source), c_j is the jth fault isolation conclusion, and **UF** is the set of unique fault isolation conclusions; and

$$f_i = \sum_{j=1}^{|\mathbf{F}|} \beta_{ij},$$

$$\beta_{ji} = \begin{cases} 1; \left(t_i \text{ does not depend on } c_j\right) \\ \quad \wedge \left(c_j \in \mathbf{UF}\right), \\ 0; \text{ otherwise} \end{cases}$$

where f_i is the number of conclusion nondependencies in t_i.

These two measures are complementary; that is, $e_i = |\mathbf{UF}| - f_i$. Hypothesis-directed search attempts to maximize the value of e_i when the test does not depend on the hypothesis and to maximize the value of f_i when the test does depend on the hypothesis. Figure 6 shows how hypothesis-directed search works. In the figure, we have masked all nonunique fault isolation conclusions (see part 3 of our series for a detailed description of uniqueness) and computed e_i and f_i for the case study. To choose a test to verify the hypothesis, we need to mask out the irrelevant members of e_i and f_i. We have done that for hypotheses of c_6 and c_{21} (examples 1 and 2, respectively). We can choose two types of tests that will verify the hypothesis in either a pass outcome or a fail outcome, but we typically choose the most robust test regardless of its outcome. In processing the data, it is important that we mask out any conclusions not to be considered and any tests not to be chosen.

For the case study, we want to confirm the outcomes of steps in a diagnostic sequence. We construct the fault tree, using weighted, entropy-directed search with the following exceptions:

- The not-needed calculation is excluded to prevent biasing the data.
- Entropy-directed search proceeds until it achieves an answer. At that point, the answer becomes the hypothesis, and we mask any tests that have already been completed. We then apply hypothesis-directed search in combination with depth-first search.
- We repeat the process for each fault isolation conclusion in the tree, thus adding one extra test to each diagnostic sequence. We can modify this procedure to include two or more extra tests by masking the extra test and reapplying the hypothesis-directed search.

Table 4 shows the fault tree optimized for minimum skill level with the addition of consistency checks (that is, hypothesis-directed search). The *inconsistency* designation occurs when a hypothesis verification test outcome is not consistent with the hypothesis. When inconsistency occurs, the technician sends the entire unit to the next repair level. Although the tree is significantly larger than the other trees, the number of steps to fault-isolate is a maximum of six and a minimum of four. In fact, each sequence has increased by only one test.

Isolation to replaceable unit in remote activities. The dependency diagram[5] shows the case study with replaceable unit boundaries. Although testing can provide isolation to the component level, in remote activities technicians might make repairs at the replaceable-unit-group level. In addition, some of the replaceable units may not be field-repairable or may be under warranty. For these occasions, we modify the termination metarules to include "Terminate when only one (or indivisible ambiguities among more than one) replaceable unit conclusion remains." During fault isolation, we need to make two other modifications. First, during the not-needed calculation, rather than considering potential new conclusion ambiguities, we consider potential new ambiguities among replaceable-unit-group conclusions. Inferring tests to be not needed requires a similar modification. When we modify the tree in Table 4 to fault-isolate replaceable units, we produce the tree in Table 5. This tree would be part of an abbreviated maintenance manual for remote operations.

Remote activities engaged in operations (verifying system availability). In deriving procedures to determine whether a system is operational or ready for delivery, it is important that we optimize the sequence of tests that leads to a *No Fault* conclusion.

Table 4. *Skill-level-optimized fault tree with consistency checks for the case study.*

Step	Test (skill level)	Previous step	Pass outcome	Fail outcome
1	t_{12} (E2)	0	Step 2	Step 13
2	t_{13} (E3)	1	Step 3	Step 8
3	t_{15} (E2)	2	Step 4	Step 5
4	t_{16} (E3)	3	c_{13}, *No Fault*	*Inconsistency*
5	t_{17} (E3)	3	Step 6	Step 7
6	t_{14} (E4)	5	c_{20}	*Inconsistency*
7	t_{11} (E7)	5	c_{19}, inu_1	*Inconsistency*
8	t_1 (E3)	2	Step 9	Step 12
9	t_4 (E4)	8	Step 10	Step 11
10	t_{11} (E7)	9	c_{16}, c_{17}, c_{18}, inu_1	*Inconsistency*
11	t_{11} (E7)	9	c_{21}	*Inconsistency*
12	t_4 (E4)	8	*Inconsistency*	c_1, c_2
13	t_1 (E3)	1	Step 14	Step 23
14	t_{18} (E3)	13	Step 15	Step 20
15	t_{10} (E6)	14	Step 16	Step 19
16	t_{11} (E7)	15	Step 17	Step 18
17	t_6 (E3)	16	c_{15}	*Inconsistency*
18	t_6 (E3)	16	c_{14}	*Inconsistency*
19	t_{11} (E7)	15	*Inconsistency*	c_{11}, c_{12}
20	t_5 (E4)	14	Step 21	Step 22
21	t_6 (E3)	20	*Inconsistency*	c_7, c_8, c_9, c_{10}
22	t_4 (E4)	20	*Inconsistency*	c_5
23	t_2 (E3)	13	Step 24	Step 29
24	int_2 (E3)	23	Step 25	Step 28
25	t_3 (E5)	24	Step 26	Step 27
26	t_5 (E4)	25	*Inconsistency*	c_6
27	t_5 (E4)	25	*Inconsistency*	c_4
28	t_3 (E5)	24	*Inconsistency*	int_2
29	int_1 (E2)	23	Step 30	Step 31
30	t_3 (E5)	29	*Inconsistency*	c_3
31	t_3 (E5)	29	*Inconsistency*	int_1

Fault trees designed to determine system operability assume a high probability of finding no faults in the system.

To derive the optimum sequence, we use a special type of hypothesis-directed search. Recall that no test depends on *No Fault*. When we examine the full dependency matrix,[5] we can easily verify that the *No Fault* row contains only zeros. For this special case, in hypothesis-directed search we completely mask the vector f and choose tests strictly by maximizing the value of e. Figure 6 shows that this maximum occurs at t_{15} or t_{16}, which is sufficient for concluding or eliminating *No Fault*. If either t_{15} or t_{16} fails, we reject the system and send it to the next level of repair. Where lower skill levels lead to uncertain test outcomes, we perform both t_{15} and t_{16}. In larger systems, we may need many tests to reach the *No Fault* conclusion. We choose each in turn, using hypothesis-directed search.

Table 5. Skill-level-optimized replaceable unit fault tree with consistency checks for the case study.

Step	Test (skill level)	Previous step	Pass outcome	Fail outcome
1	t_{12} (E2)	0	Step 2	Step 13
2	t_{13} (E3)	1	Step 3	Step 8
3	t_{15} (E2)	2	Step 4	Step 5
4	t_{16} (E3)	3	ru_6, *No Fault*	*Inconsistency*
5	t_{17} (E3)	3	Step 6	Step 7
6	t_{14} (E4)	5	ru_8	*Inconsistency*
7	t_{11} (E7)	5	ru_8, inu_1	*Inconsistency*
8	t_1 (E3)	2	Step 9	Step 12
9	t_4 (E4)	8	Step 10	Step 11
10	t_{11} (E7)	9	ru_7, ru_8, inu_2	*Inconsistency*
11	t_{11} (E7)	9	ru_2	*Inconsistency*
12	t_4 (E4)	8	*Inconsistency*	ru_1, ru_2
13	t_{18} (E3)	1	Step 14	Step 17
14	t_{10} (E6)	13	Step 15	Step 16
15	t_{11} (E7)	14	ru_6	ru_6
16	t_{11} (E7)	14	*Inconsistency*	ru_5, ru_6
17	t_2 (E3)	13	Step 18	Step 25
18	t_3 (E5)	17	Step 19	Step 22
19	t_5 (E4)	18	Step 20	Step 21
20	t_6 (E3)	19	*Inconsistency*	ru_4, ru_5
21	t_6 (E3)	19	*Inconsistency*	ru_3
22	int_2 (E3)	18	Step 23	Step 24
23	t_5 (E4)	22	*Inconsistency*	ru_3
24	t_5 (E4)	22	*Inconsistency*	int_2
25	int_1 (E2)	17	Step 26	Step 27
26	t_3 (E5)	25	*Inconsistency*	ru_1
27	t_3 (E5)	25	*Inconsistency*	int_1

Table 6. Test groups for the case study.

Group number	Group members	Comments
1	int_2, int_1	Perform first in listed sequence
2	t_4, t_{10}, t_{11}	Penalize 3.0 time units
3	t_1, t_2, t_3, t_5	
4	t_6, t_7, t_8, t_9, t_{18}, t_{12}, t_{13}, t_{14}	
5	t_{15}	Individual test, ungrouped
6	t_{16}	Individual test, ungrouped
7	t_{17}	Individual test, ungrouped

Diagnosis in the shop environment

As we discussed earlier, in several instances a unit is not repaired in the field. In a shop environment, different considerations may drive the development of fault trees. For example, although all necessary test resources are available in the shop, for work flow efficiency all tests requiring a certain skill level or common test equipment should be contiguous in the fault tree. A collection of tests with common factors is called a test group. Some test groups are easier to perform than others, and some may have penalties associated with their execution. Finally, some test choices are just too critical to be left to the computer.

Handling test groups. In the case study, we have defined four test groups, shown in Table 6. The tests in each group have some logical relationship to each other, such as requiring the same test equipment. Under ordinary fault isolation conditions, we would like to complete the tests in one group before proceeding to the next group. We accomplish this as follows:

- When we are not currently in a test group, we choose a test by using the normal optimization process.
- When we are currently in a group and have not completed that group, we restrict the test choice to the available tests in the group.
- When we complete a test group, we then choose from among all remaining tests or test groups.

Not all tests are as desirable as others. For example, in the inertial navigation subsystem of the B-747 aircraft, a set of readable output jacks is located behind a skin panel fastened with 43 flush-mounted screws. When test sequences were developed for this subsystem, tests using the output jacks were placed in a test group. Accessing tests in that test group resulted in a time loss because the skin panel first had to be removed. Ac-

cordingly, the test group would receive a time penalty weighting that would cause tests in the group to be delayed and in some cases avoided. Assuming the case study system requires a similar access panel to perform tests in group 2, we have placed a penalty of 3.0 time units on that group (Table 6).

Sequencing. In some cases, the computer should not determine the test sequence. For example, we should perform tests designated "safe-to-turn-on" first, but optimization may fail to provide this sequencing. In the case study, we can assume that if we begin testing the system and the inputs are not present, we may damage the equipment. Accordingly, we have designated a test group that includes the two inputs and directed that it must be completed first. This procedure is called *group sequencing*. We can sequence all the groups in turn if we wish. In addition, there is a chance that testing int_1 first will cause an electrostatic discharge if int_2 is not valid. Therefore, within the group we have sequenced the tests so that int_2 is chosen first. Sequencing of this type must be consistent with associated inferences (with a couple of exceptions).

Multiple failures. Growth in computational complexity prohibits the development of comprehensive multiple-failure fault trees. Nevertheless, in several cases, multiple failures are a significant problem in developing diagnostic strategies. For the case study, let us consider two such multiple failures. In part 4 of this series,[4] we discovered that the potential for false indication due to a multiple failure was present in the system. That is, a combined failure of c_1 and c_5 or a combined failure of c_2 and c_5 could result in a diagnosis of c_6. Suppose engineering analysis has revealed that failure of c_1 and c_5 is indeed possible under certain conditions. We wish to include this multiple failure in the fault tree, and we call the associated group

Conclusions	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}	t_{18}	int_1	int_2	t_x
int_1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
int_2	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0
c_1	1	0	0	1	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0
c_2	1	0	0	1	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0
c_3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
c_4	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
c_5	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
c_6	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
c_7	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
c_8	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
c_9	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
c_{10}	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
c_{11}	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
c_{12}	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
c_{13}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
c_{14}	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	0	0
c_{15}	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	0	0	0	0
c_{16}	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0
c_{17}	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0
c_{18}	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0
c_{19}	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0
c_{20}	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
c_{21}	0	0	0	1	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0
inu_1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0
inu_2	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0
No Fault	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Failure 1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1
Failure 2	0	0	0	1	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	0	0

Tests

Figure 7. *Multiple-failure mapping analysis for case study.*

Failure 1. In addition, suppose that engineering analysis has determined that failure of c_{14} may actually trigger failure of c_{21} and vice versa. We include this multiple failure as *Failure 2*. Although generating fault trees that include all multiple failures is intractable, we can include these specific multiple failures in the trees and in the testability analysis by mapping the multiple failures to single conclusions in the model.

We obtain the conclusion *Failure 1* by combining all the elements of its two constituent parts (c_1 and c_5). In this case, testability analysis shows an ambiguity between the *Failure 1* conclusion and the c_6 conclusion, which we should have anticipated from the previous analysis of false indications. To compensate, we add a special test, t_x, which looks specifically for the multiple failure and nothing else. Analysis shows that *Failure 2* is not a problem and is uniquely isolatable with the current test set.

Figure 7 shows the complete remapping of the test-to-conclusion matrix for the two multiple failures, including t_x. Once we have mapped the additional elements into the model, developing fault trees is straightforward. Table 7 shows the tree that includes these multiple failures. With all factors used, the tree requires between one and seven tests for resolution and an expected time

Table 7. Tailored fault tree table with test groups and test sequencing for the case study.

Step	Test	Previous step	Pass outcome	Fail outcome
1	int_2	0	Step 2	int_2
2	int_1	1	Step 3	int_1
3	t_{18}	2	Step 4	Step 13
4	t_{12}	3	Step 5	Step 10
5	t_{13}	4	Step 6	Step 8
6	t_{14}	5	Step 7	c_{19}, inu_1
7	t_{15}	6	c_{13}, *No Fault*	c_{20}
8	t_4	5	c_{16}, c_{17}, c_{18}, inu_2	Step 9
9	t_1	8	c_{21}	c_1, c_2
10	t_{11}	4	c_{15}	Step 11
11	t_4	10	Step 12	*Failure 2*
12	t_{10}	11	c_{14}	c_{11}, c_{12}
13	t_1	3	Step 14	Step 15
14	t_5	13	c_7, c_8, c_9, c_{10}	c_5
15	t_3	13	Step 16	Step 17
16	t_x	15	c_6	*Failure 1*
17	t_2	15	c_4	c_3

of 7.55 time units (including the penalty of 3.0 time units in seven of the 18 isolation sequences).

Special conditions for fault isolation

In some situations, we may wish to override the inferences drawn in the inference engine, modify the way we provide information, or tailor the rules to terminate isolation. The following paragraphs describe five basic types of overrides, or special considerations.

A priori information (prior inference). Before we begin diagnosis, we may have information about the failure. For example, a light indicating trouble with the oil may be illuminated, or certain components may have just been checked and can be assumed good. Incorporating this information before fault isolation can reduce the search space and improve efficiency. Any information available should be processed by the inference engine before we develop fault trees. Let us assume that the system is being evaluated at the depot, using calibrated input sources, and that these inputs are good. This results in a fault tree that does not include inputs either as tests or as conclusions.

Suppressed test inference. We saw earlier that one way to address the false-alarm problem is to assume the diagnosis was incorrect and choose additional tests, using hypothesis-directed search. A second way is to assume some testing is wrong. Because the outcomes of some tests may be determined by other tests, we may wish to force execution of certain tests, regardless of whether or not they can be inferred. These tests may be the most accurate, the most reliable, or the easiest to perform properly. We can place them in the tree, or we can suppress inference by the inference engine (thus forcing them to be tested).

Linkages. Dependencies specified in the model provide certain direct linkages for inference. The model also includes the capability to specify several types of linkages for inference outside the dependency matrix representation. Such linkages include:

- A fail test outcome, triggering pass inferences (for example, a light that indicates a failure also indicates a good light bulb and good voltage to the light)
- A pass test outcome, triggering fail inferences
- A fail test outcome, triggering an untestable test or tests (for example, unsafe-to-test)
- A pass test outcome, triggering an untestable test or tests

We can derive these linkages directly from the inference metarules. Let us assume that a fail outcome of t_4 renders t_1 unsafe to test. This combination is the most frequently encountered and is closely related to the safe-to-turn-on tests (normally handled as a test group and sequenced first in the diagnostic process). As a result, if t_4 is evaluated, the subtree corresponding to t_4's failing will not include t_1 because the linkage causes t_1 to be inferred untestable.

Intermediate conclusions. Often during isolation, we wish to note progress or specific conclusions as we proceed. These intermediate conclusions exist in the model as tests. These tests are immediately declared untestable, but if another value is assigned (through inference), then that value is reported. For example, we can define a test that depends on all conclusions designated c_1 to c_{10}, a test that depends on all conclusions designated c_{11} to c_{21}, and a test that depends on all the inputs. We would then designate these three tests intermediate conclusions. The intermediate conclusions, then, would provide information about two halves of the system and about all the inputs. An intermediate conclusion

means several things to the inference engine:

- The element is considered untestable.
- The element is not to be counted during any optimization.
- The element is to be announced whenever an inference rule determines its state.

Any combination of elements can appear as an intermediate conclusion. In a complex avionics system, we can consider all warranty items with an intermediate conclusion. This construct provides a means for arbitrarily including inferable information in the model without biasing the optimization process.

Multiple-outcome tests. So far, we have assumed that test outcomes are binary; that is, tests either pass or fail. However, entropy-directed depth-first search is fully capable of analyzing tests with multiple outcomes. Previously, the backup from a pass outcome led to processing a fail outcome, and the backup from a failed outcome led to another backup in the tree. When multiple outcomes are present, we stay with the individual node until all outcomes are exhausted. That is, on first arrival at a node, we proceed with the choice and assign the first outcome. When we arrive at an answer, we back up and examine the node, choose the next outcome (if available) in line, or back up further.

In the case study, we can assume that in field situations certain tests may not be performable (because of equipment or personnel shortage) and certain tests may be potentially untestable. Thus, the tests have three outcomes: pass, fail, and unknown. If outcomes other than pass and fail are used in a model, then we may have to modify the inference metarules and extend the matrix beyond binary. Otherwise, the optimization process remains unchanged.

IN OUR DISCUSSION OF THE CASE STUDY, we derived five different fault trees for different circumstances. The fault tree shown in Table 7 provides diagnostic sequences to be used in the shop; those in Tables 2, 3, and 4 would be used in the operational environment; and the fault tree in Table 5 would be used with a special checkout procedure in operational units engaged in remote operations.

A portion of fault tree development should be performed in concert with the testability analyses described in parts 3 and 4. Incorporation of false-alarm checks and multiple-failure diagnosis are two examples of the interaction between fault isolation and testability analysis. The modeling process allows us to answer what-if questions and develop tailored products such as technical manuals for integrated diagnostics.

This article concludes our presentation of the basic concepts of the integrated diagnostics modeling approach. We refer the reader to papers on several related advanced topics not discussed here:

- Performing interactive fault isolation: using the model to diagnose in a dynamically changing environment.[14]
- Learning from actual isolations: modifying time, failure frequency, and other data, as well as learning relationships from diagnostic instances.[15,16]
- Reasoning under uncertainty: what to do when outcomes are not exact, technicians' skill levels are in doubt, or other uncertainty factors are present.[17,18]
- Partitioning large systems and models: Often, large systems translate to slow execution in a dynamic environment. Partitioning breaks the system into several submodels.[19]
- Developing an architecture for intelligent, learning, adaptable, and modifiable automatic test equipment.[20]
- Tying system testing to logistics: providing for logistics feedback, including computer-aided logistics support (CALS).[21]
- Temporal factors: applying point-based and interval-based time constraints.[22]

Developing fault trees for actual, practical diagnosis in an integrated diagnostic environment is a complex process. Considering various overrides and constraints allows computation of a virtually infinite number of fault trees for any complex system. Real fault trees may simultaneously invoke several of the factors we have discussed. In fact, the development process most likely will require iterative design of the fault trees. Incorporating constraints and overrides in the model and algorithms will be necessary to consistently generate practical diagnostic strategies that do not require manual manipulation by an expert. D&T

Acknowledgments

We thank the following people, without whom we could not have produced this series of articles: Colin Maunder, Leonard Haynes, Sharon Goodall, Les Orlidge, Sheryl Sieracki, Elizabeth Reed, Ken Wagner, and the editors of *IEEE Design & Test*. We also thank Jon Agre, Brian Kelley, Jerry Graham, Steve Troy, Brian Pickerall, and the many others who helped us to develop concepts or to see the difference between theory and practice.

References

1. W.R. Simpson and J.W. Sheppard, "System Complexity and Integrated Diagnostics," *IEEE Design & Test*, Vol. 8, No. 3, Sept. 1991, pp. 16-30.
2. J.W. Sheppard and W.R. Simpson, "A Mathematical Model for Integrated Diagnostics," *IEEE D&T*, Vol. 8, No. 4, Dec. 1991, pp. 25-38.
3. W.R. Simpson and J.W. Sheppard, "System Testability Assessment for Integrated Diagnostics," *IEEE D&T*, Vol. 9, No. 1, Mar. 1992, pp. 40-54.

4. J.W. Sheppard and W.R. Simpson, "Applying Testability Analysis for Integrated Diagnostics," *IEEE D&T*, Vol. 9, No. 3, Sept. 1992, pp. 65-78.
5. W.R. Simpson and J.W. Sheppard, "Fault Isolation in an Integrated Diagnostic Environment," *IEEE D&T*, Vol. 10, No. 1, Mar. 1993, pp. 52-66.
6. C.E. Shannon, "A Mathematical Theory of Communications," *Bell Systems Technical J.*, Vol. 27, 1984, pp. 379-423.
7. L. Breiman et al., *Classification and Regression Trees*, Wadsworth, Belmont, Calif., 1984.
8. J.R. Quinlan, "Induction of Decision Trees," *Machine Learning*, Vol. 1, 1986, pp. 81-106.
9. F.I. Dretske, *Knowledge and the Flow of Information*, MIT Press, Cambridge, Mass., 1982.
10. S.R. Troy, *Cost as a Matrix Input to STAMP*, STAMP Tech. Note 266, Arinc Research Corp., Annapolis, Md., 1990.
11. J.G. Malcom, "BIT False Alarms: An Important Factor in Operational Readiness," *Proc. Annual Reliability and Maintainability Symp.*, IEEE, Piscataway, N.J., 1982, p. 206.
12. *Design Guide, Built-in Test (BIT) and Built-in Test Equipment (BITE) for Army Missile Systems*, Report TR-RL-CR-81-4, Sperry Corp., Minneapolis, Minn., 1981.
13. W.R. Simpson and J.W. Sheppard, "Analysis of False Alarms During System Design," *Proc. IEEE Nat'l Aerospace Electronics Conf.*, IEEE, Piscataway, N.J., 1992, pp. 657-661.
14. J.W. Sheppard and W.R. Simpson, "Incorporating Model-Based Reasoning in Interactive Maintenance Aids," *Proc. IEEE Nat'l Aerospace Electronics Conf.*, IEEE, Piscataway, N.J., 1990, pp. 1238-1243.
15. J.W. Sheppard and W.R. Simpson, "Elements of Machine Learning in a Field Diagnostic Maintenance Aid," *Proc. Artificial Intelligence Applications for Acquisition Management, Logistics Management, and Personnel Management Conf.*, American Defense Preparedness Assoc., Williamsburg, Va., 1992, pp. 7-13.
16. J.W. Sheppard, "Explanation Based Learning with Diagnostic Models," *IEEE Autotestcon Conf. Record*, 1992, pp. 159-167.
17. J.W. Sheppard and W.R. Simpson, "Uncertainty Computations in Model Based Diagnostics," *IEEE Autotestcon Conf. Record*, 1991, pp. 233-242.
18. J.W. Sheppard and W.R. Simpson, "A Neural Network for Evaluating Diagnostic Evidence," *Proc. IEEE Nat'l Aerospace Electronics Conf.*, 1991, pp. 717-724.
19. W.R. Simpson and J.W. Sheppard, "Partitioning Large Diagnostic Problems," *IEEE Autotestcon Conf. Record*, 1991, pp. 329-327.
20. W.R. Simpson and J.W. Sheppard, "An Intelligent Approach to Automatic Test Equipment," *Proc. Int'l Test Conf.*, IEEE Computer Society Press, Los Alamitos, Calif., 1991, pp. 419-426.
21. A.B. Blair, J.W. Sheppard, and W.R. Simpson, "A Partnership for Systems Support: Artificially Intelligent Maintenance Aids and CALS," *Logistics Spectrum, J. of Society of Logistics Engineers*, Vol. 26, No. 3, Summer 1992, pp. 19-26.
22. J.W. Sheppard and W.R. Simpson, "Fault Diagnosis Under Temporal Constraints," *IEEE Autotestcon Conf. Record*, 1992, pp. 151-159.

John W. Sheppard is a principal research analyst in the Advanced Research and Development Group at Arinc Research Corporation. He is also a PhD candidate in computer science at Johns Hopkins University. His research interests include applying AI techniques to fault diagnosis, machine learning, neural networks, and knowledge representation. He was a principal developer of Pointer, an intelligent, interactive maintenance aid, and assisted in the development of a prototype expert system that diagnoses system failures and reconfigures the system to maintain functioning. He holds a BS from Southern Methodist University and an MS from Johns Hopkins University, both in computer science.

William R. Simpson, a research fellow in the Advanced Research and Development Group at Arinc Research Corporation, works on 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 the Pointer 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.

Direct questions or comments on this article to the authors at Arinc Research Corp., Advanced R&D Group, 2551 Riva Rd., Annapolis, MD 21401; sheppard@csjhu.edu or william_r_simpson@mcimail.com.