A Mathematical Model for Integrated Diagnostics

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

In part 2 of a series on integrated diagnostics, the authors expand on the form of the information flow model they introduced in the first article.[1] Compiling the model requires three algorithms for determining higher order relationships. One of these, the algorithm for computing logical closure, helps to simplify the modeling task. The authors also introduce a hypothetical antitank missile launcher to illustrate concepts and computations presented in this and subsequent articles in the series.

Since its introduction in the 1980s, testability analysis has emerged as a significant engineering discipline. Previously, such analysis began only after systems were fielded, and the results often exhibited poor field maintainability.[2,3] Buyers and users of complex systems are now demanding more precise analysis of field maintainability during the design phase of these systems. In response to such demands, a number of companies and universities have developed approaches to address the issue of field maintainability. Many of these approaches use model-based reasoning to provide design for testability analysis and diagnosis.[4-7]

System testability means the ability to test a system. In this series of articles, we concentrate on our ability to diagnose failures as part of an overall integrated maintenance architecture. We agree with the definition of testability "... as 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."[8]

Testability is a design concept by which we gauge our success in achieving design goals for field maintenance. It comprises several issues concerned with the various aspects of maintaining complex systems. To address several testability issues mathematically, we developed the information flow model.

In the first article[1] of this series, we provided an overview of the problem of analyzing testability and conducting diagnosis for complex systems. We now expand on the form of the information flow model that was introduced in the first article and has been used successfully in several types of tools.[9,10]

The major elements of the information-flow model include graph-based and logic-based representations, groupings, and multiple-conclusion mappings. To assist modeling, our information flow model enables an analyst to specify a simplified form of the system model, which the testability software later compiles to facilitate analysis of system testability. Compiling the model requires three algorithms for determining higher order relationships. This form of the system model provides a structure for analyzing testability and diagnosis, which we will discuss in future articles in this series.

0740-7475/91/0012-0025$01.00 © 1991 IEEE

Case study

We will use a hypothetical antitank missile launcher to illustrate the concepts and computations described in these articles. Tables 1 and 2 provide test and failure mode data for the case study. We derived the case study from an actual missile system, modifying it extensively to illustrate certain mathematical principles. As a result, although the data represent an actual problem, the system may deviate significantly from what may be encountered in a real missile system.

The hypothetical missile launcher consists of a tripod, a gunner's optical sight; a launch tube; a traversing unit to which the tripod, launch tube, and optical sight attach; and an electronic guidance computer. The missile contains two solid-propellant motors. The launch motor ejects the missile from the launch tube and is burned out by the time the missile has left the tube. Only after the missile has flown several meters does the flight motor ignite, so no protection is required for the gunner.

After the missile leaves the launch tube, a light source in the tail comes on so the optical sensor on the launcher can track the missile along its flight path. The light source is sufficiently strong to allow automatic guidance to the maximum range of the missile under all conditions in which the missile is visible to the gunner.

Table 1. Tests in the case study.

Available tests	Label*	Time**	Skill level†
Fire command signal	int_1	1.00	E2
Reticle position tracker	int_2	1.00	E3
One-two fidelity signal	t_1	2.00	E3
Readiness output signal	t_2	2.20	E3
Boundary parameter signal	t_3	2.40	E5
Summary safe/arm signal	t_4	1.50	E4
Combined fidelity signal	t_5	1.30	E4
Error signal	t_6	3.00	E3
Course correction signal	t_7	1.00	E5
Command corrector signal	t_8	2.00	E3
Command response signal	t_9	0.50	E4
Track signal	t_{10}	0.60	E6
Power ready signal	t_{11}	0.10	E7
Power ready/enable signal	t_{12}	0.90	E2
Bounded error signal	t_{13}	1.20	E3
Launcher enable signal	t_{14}	1.60	E4
Launch track generator signal	t_{15}	1.50	E2
Launch track evaluation signal	t_{16}	2.00	E3
Launcher ready signal	t_{17}	1.50	E3
Stabilized loop signal output	t_{18}	0.30	E3

* *int corresponds to a testable input, including both the test and conclusion element, and t corresponds to a test.*
** *Units are not significant as long as they are consistent.*
† *Skill level corresponds to enlisted rank in the US military services. We assume a linear correspondence between rank and skill level.*

Information flow model

The structure of the information flow model facilitates our ability to formulate testability measures. An information flow model has two primitive elements: *tests* and *fault-isolation conclusions.* Tests include any source of information that can be used to determine the health of a system.

Fault-isolation conclusions include failures of functionality, specific nonhardware failures (such as bus timing), specific multiple failures, and the absence of a failure indication. The information obtained may be a consequence of the system operation or a response to a test stimulus. Thus, we include observable symptoms of failure processes in the information flow model as tests. Doing this allows us to analyze situations that involve information sources other than formally defined tests. The purpose of our model, of course, is to combine these information sources (tests) to derive conclusions about the system being diagnosed.

The basic representation of the information flow model includes both a dependency representation and a logical representation of the system being analyzed. In addition, the information-flow model includes the definition of groups of logically related tests and conclusions. In this representation, we define logical values for tests and fault-isolation conclusions. Specifically, if a test fails, it is true; if a test passes, it is false. An asserted conclusion is true; a conclusion eliminated from consideration is false.

Test information

The procedures in the maintenance manual for the antitank missile launcher provide detailed stimulus and response data for each of the tests. The following represents the information obtained from the maintenance procedures:

Table 2. Conclusions in the case study.

Failed element	Label*	Rate**	Replaceable unit †
Fire command signal	int_1	10	0
Reticle position tracker	int_2	10	0
Command override	inu_1	10	0
Override enable	inu_2	10	0
Sight activation function	c_1	5	Assembly 1 (ru_1)
Safe/arm determination	c_2	5	Assembly 2 (ru_2)
Launcher ready evaluator	c_3	100	Assembly 1 (ru_1)
Boundary check	c_4	100	Assembly 3 (ru_3
Parameter fidelity	c_5	100	Assembly 3 (ru_3)
Parameter fidelity backup	c_6	100	Assembly 3 (ru_3)
Error evaluator	c_7	5	Assembly 4 (ru_4)
Error corrector	c_8	5	Assembly 4 (ru_4)
Command signal evaluator	c_9	5	Assembly 5 (ru_5)
Response generator	c_{10}	5	Assembly 5 (ru_5)
Target tracker	c_{11}	5	Assembly 5 (ru_5)
Command to track comparator	c_{12}	5	Assembly 6 (ru_6)
Guidance output	c_{13}	5	Assembly 6 (ru_6)
Launcher power supply (battery)	c_{14}	100	Assembly 6 (ru_6)
Launcher power enable	c_{15}	100	Assembly 6 (ru_6)
Launcher power signal conditioner	c_{16}	5	Assembly 7 (ru_7)
Cross-check override	c_{17}	5	Assembly 7 (ru_7)
Error signal boundary check	c_{18}	5	Assembly 8 (ru_8)
Launcher full ready function	c_{19}	5	Assembly 8 (ru_8)
Launch command function	c_{20}	100	Assembly 8 (ru_8)
Fire ready activation function	c_{21}	100	Assembly 2 (ru_2)

* *int corresponds to a testable input, including both the test and conclusion element, inu corresponds to an untestable input, and c corresponds to a conclusion.*

** *Units are not significant as long as they are consistent.*

† *Replaceable units are designated in accordance with the definition of groups given in the section "Representing group constructs." The system inputs are not listed as members of replaceable unit groups, although they could be.*

- *One-two fidelity signal (t_1).* This signal examines both the safe/arm determination (c_2) and the sight activation function (c_1). If either element is faulty or improper, this signal detects the problem. In addition, a faulty fire command signal (int_1) or a faulty boundary parameter signal (t_3) adversely affects the one-two fidelity signal. This signal also detects failure of the parameter fidelity backup unit (c_6).
- *Bounded error signal (t_{13}).* Before the launch circuitry issues final launch and guidance instructions, it performs a final evaluation of all signals. The bounded error signal examines the signals produced by the launcher power signal conditioner (c_{16}), the cross-check override (c_{17}), the error signal boundary check (c_{18}), and the override enable (inu_2). The bounded error signal also cross-checks the stabilized loop signal output (t_{18}), the power ready/enable signal (t_{12}), and the summary safe/arm signal (t_4). If any of these units has failed, the bounded error signal indicates a failure.

Directed graph

In using a directed graph to model information flow, the primitive elements (that is, tests and conclusions) are the vertices, and dependency relationships between the primitive elements are the edges. More formally, let the set of vertices **V** equal $\mathbf{I} \cup \mathbf{F}$, where **I** represents the set of information sources (tests), and **F** represents the set of fault-isolation conclusions. Let the set of edges **E** equal the set of first-order dependency relationships between the vertices in **V**. By *first order*, we mean the direct relationships between vertices without forward or backward chaining (that is, the set of paths of length equal to 1 from a test back to a test or conclusion). Let $\mathbf{D} = (\mathbf{V}, \mathbf{E})$ be an adjacency matrix that represents the dependency graph of the system being analyzed.

We can determine *higher order* dependency relationships (designated by $\hat{\mathbf{E}}$)—that is, the set of paths of length greater than or equal to 1—for each test by using several algorithms. These algorithms provide information equivalent to that provided by traditional forward- and backward-chaining algorithms in a rule-based inference system. Let $\hat{\mathbf{D}} = (\mathbf{V}, \hat{\mathbf{E}})$ represent the higher order dependency graph of the system being analyzed.

We store both the first-order and the higher order dependency graphs as bit-adjacency matrices, where each cell in the matrix uses only one bit of memory. The bit matrix representation is compact and requires only $\theta(n^2)$ bits for storage. Here, n is the number of elements in the matrix, corresponding to the sum of the testable ele-

ments and the conclusion elements plus 1 (for a special conclusion referred to as *No Fault*). A testable input comprises two elements (test and conclusion) under this formulation. We separate tests and conclusions in the dependency matrix to simplify several calculations and analyses.

Recall that **I** represents the set of information sources, which also includes testable inputs. And **F** represents the set of fault-isolation conclusions, which also includes testable and untestable inputs, multiple failures included in the dependency graph, and *No Fault*. (An input is any signal coming in from outside the system; a testable input is an input that we can independently test.) The top half of the dependency matrix represents the test-to-test dependency relationships and is of size $|\mathbf{I}| * |\mathbf{I}|$. (Given a set **X**, $|\mathbf{X}|$ represents the cardinality of **X**, that is, the number of members of set **X**.) The bottom half of the dependency matrix represents the test-to-conclusion dependency relationships and is of size $|\mathbf{I}| * |\mathbf{F}|$.

On the basis of the test descriptions of the case study, we can determine the first-order test dependencies for t_1 and t_{13} as follows:

- t_1 depends on int_1, t_3, c_1, c_2, and c_6, and
- t_{13} depends on inu_2, t_4, t_{12}, t_{18}, c_{16}, c_{17}, and c_{18}.

Figure 1 is a dependency diagram for the missile launcher. Figure 2 shows all the first-order dependencies (indicated by "f") mapped into a matrix representation. By definition, no test has a dependency on *No Fault*, allowing us to make the closed-world assumption. In this type of model development, we generally mark columns of the matrix to represent test dependency lists, as shown in Figure 2.

Representing logical constructs

The matrix orientation forces a logical interpretation of the information in the rows and columns. We can say that if a given conclusion is true, all the tests that depend on the conclusion are also true. In other words, the tests that depend on the conclusion (represented by row elements) will detect the failure. This is also true in the event that we know that a test has a failed outcome. In this case, all tests that depend on the failed test must also fail. In addition, the columns of the dependency matrix provide information concerning the possible cause of a failure. If a test passes, all the elements in the corresponding column (both tests and conclusions) must also pass.

The graphical form of representation is limited. That is, we may want the logical constructs given in Equations 1 through 3 (which correspond to Equations 4 through 6 in part 1[1]):

$$conclusion_i \supset \sum_j test_j, \tag{1}$$

where Σ represents disjunction, given that $test_j$ depends on $conclusion_i$;

$$test_i \supset \sum_j test_j, \tag{2}$$

given that $test_j$ depends on $test_i$; and

$$\neg test_i \supset \left(\sum_j \neg test_j\right) \vee \left(\sum_k \neg conclusion_k\right) \tag{3}$$

given that $test_i$ depends on $test_j$, $conclusion_k$.

The matrix formulation does not directly handle Equations 1 through 3. Equations 1 and 2 result when a fault *might* lead to the failure of one or more tests and the failure of a test *might* lead to the failure of one or more other tests. We call the set of tests that may fail a test-disjunct set. Equation 3 provides for the inclusion of multiple conclusions in the model. We derived the basic formulation to limit the combinatorial growth of the search space. However, it is important to include the logical relationships because they are a part of real systems. Because the matrix representation does not directly support these three constructs, we add special elements to the model to overcome the limitation.

To incorporate the test-disjunct sets of Equations 1 and 2, we create a separate test element that is the logical OR of the matrix columns corresponding to the individual tests within the test-disjunct set:

$$\hat{\mathbf{D}}_i = \sum_{\forall t_j \in T_i} \hat{\mathbf{D}}_j, \tag{4}$$

where $\hat{\mathbf{D}}_i$ represents the ith column of the higher order dependency graph, t_j is a test that belongs to the test-disjunct set, and T_i is the set of tests in the ith test-disjunct set to be represented in the matrix.

For the case study, Equation 1 corresponds to mapping a construct such as the following. If c_{17} fails, either t_1 or t_{13} or both will have a bad test outcome. This logic is handled by an additional element t_1-or-t_{13}, in the matrix form that depends on t_1, t_{13}, and c_{17}.

Equation 2 corresponds to mapping a construct such as this. If t_9 has a bad test outcome, either t_1 or t_{13} or both will have a bad test outcome. This logic is handled by an additional element t_1-or-t_{13}, in the matrix form that depends on t_1, t_{13}, and t_9.

If both constructs are present, the compound representation of t_1-or-t_{13}, depending on t_1, t_{13}, t_9, and c_{17}, will suffice.

To incorporate the construct given by Equation 3, we create a separate conclusion that is the logical And of the matrix rows corresponding to the individual conclusions within a multiple conclusion group. We discuss this mapping in detail in Part 1 of this series.[1]

Representing group constructs

A *group* is a collection of similar elements that all have a common aspect significant to diagnosis. For example, a test group may have a collection of tests that all require the same equipment. A replaceable unit group contains all of the conclusions indicative of a failure in one piece of hardware.

The basic construct used to represent a

group within this formulation is the *set*. Groups, which are represented outside the matrix, affect several analyses of testability and are represented using characteristic vectors. In particular,

$$\mathbf{G}_i^{\mathbf{S}j} = \begin{cases} j; i \in \mathbf{S}_j \\ 0; \text{ otherwise} \end{cases} \quad (5)$$

Here, $\mathbf{G}_i^{\mathbf{S}j}$ is the *i*th element of the characteristic vector for the *j*th group of type **S** corresponding to the *i*th conclusion in the model. We may interpret an element assigned a value of 0 in either one of two ways

- That element is considered to be a

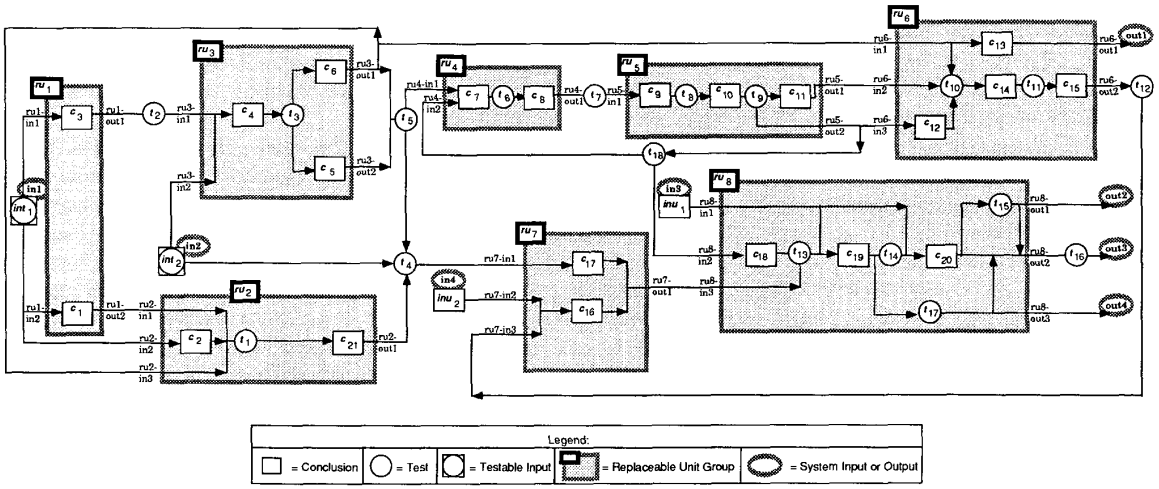

Figure 1. *Dependency diagram for the case study.*

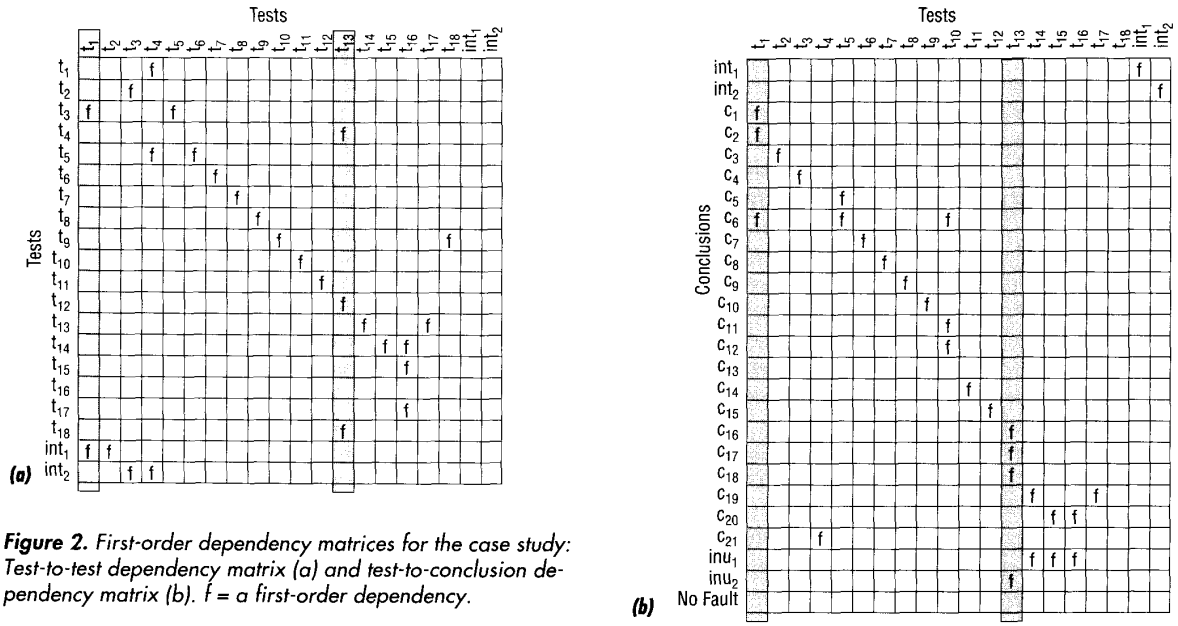

Figure 2. *First-order dependency matrices for the case study: Test-to-test dependency matrix (a) and test-to-conclusion dependency matrix (b). f = a first-order dependency.*

group whose only member is that element
- That element is not a member of any group of this type

The appropriate interpretation is based on the group type. This formulation uses several group types. One is a *replaceable unit group* (a group of fault-isolation conclusions to be treated as a single isolatable element). Another is a *test group* (a group of logically related tests to be performed together), and a third is a *failure group* (a group of fault-isolation conclusions to be treated as a conclusion only when all of its members are true). These groups, which the analyst provides, represent examples of several types of groups handled by this formulation. The analyses we describe in this series of articles identify several additional groups, including ambiguity groups, redundant test groups, feedback groups, and hidden-failure groups. For the case study, the following groups may be assigned (see Figure 1):

- *Replaceable unit group.* For example, replaceable unit group ru_1 comprises c_1 and c_3.
- *Test group.* An example is t_4, t_{10}, and t_{11}—a group of tests that require an oscilloscope.
- *Failure group.* An example failure group consists of c_1 and c_5, which are expected to fail at the same time.

Processing a model

We consider the information flow model to be the knowledge base of the system to be analyzed and evaluated. An analyst prepares a simplified information flow model that consists of first-order dependencies, group specifications, and other information specific to the system being analyzed. We call the process of converting the simplified information flow model into a form suitable for analysis *knowledge-base compilation.*

Compilation is performed in several steps to determine all the implications of a test. The first such analysis is based on the *transitivity* of logical implication. When A implies B and B implies C, then A implies C. This is the transitive property. The process of mapping these implications in a graph representation is called *transitive closure.*

Because the graph representation captures the system topology, we can make a number of calculations before proceeding further, including feedback analysis and consistency cross-checking of the manner in which conditional elements are handled.

The internal inference rules are cross-checked for additional implications in a process called *logical closure.* When logical closure provides a new implication, it must again be checked for transitivity effects—a process called *incremental closure.* When processing is complete, the total implications available from each outcome of each test are available in the matrix representation. Compiling a knowledge base includes performing transitive closure, feedback analysis, conditional cross-checking, logical closure, and incremental closure.

The process of developing information-flow models includes specifying the primitive elements of the model, determining the dependency relationships between the primitive elements, characterizing the types of tests, providing appropriate weighting criteria for fault isolation, and specifying appropriate groupings and test sequences.[1] The most important step in modeling is determining the dependencies. To simplify this process, the analyst enters first-order dependencies for each test in the model. We determine a higher order dependency, on the other hand, by applying the transitive property to the first-order dependencies in the flow graph.

The analyst determines the first-order dependencies for each test in the model. The analyst traces the information flow back from the test until another test or an input is encountered or no other elements are encountered. The testability analysis considers all conclusions on the resulting path as dependencies, as well as the test or input that lies on the path. Because multiple paths may feed a test, the analysis must also consider all paths flowing to the test.

Required process steps

Once the analyst develops an information flow model of a system, the testability software preprocesses the model to determine the higher order dependencies.

Transitive closure. The first step in preprocessing is to compute the transitive closure of the dependencies in the model. Several algorithms exist for this computation.[11] For our implementation, we selected Warshall's algorithm for bit matrices because the system stores the dependency information in a binary matrix. Our matrix has two portions corresponding to test-to-test relationships and test-to-conclusion relationships. We modified the algorithm slightly to complete the closure in the test-to-conclusion portion of the matrix (see Figure 3). Because we have a bit matrix, we can achieve a great deal of machine efficiency by storing several bits in a word of memory and then applying logical operations to the words instead of the bits.

Feedback analysis. Following transitive closure, a feedback analysis is performed to identify topological circularities in the model. Such circularities may result from physical feedback, information-flow feedback, or modeling error. There may be additional circularities following logical closure; these circularities are identified as additional test redundancies. The software identifies topological circularity for a test t_i if and only if t_i depends on itself.

To assign each test to its feedback group, let $\tilde{\mathbf{D}}'$ represent a dependency matrix following transitive closure but before logical closure, and let $\tilde{\mathbf{D}}'^{c}_{i}$ and $\tilde{\mathbf{D}}'_{j}$ be vectors such that

$$\tilde{\mathbf{D}}'^{c}_{i} = \left\{\tilde{\mathbf{D}}'_{ij} \middle| \forall j\right\} \left(\text{the } i\text{th-column vector of } \tilde{\mathbf{D}}'\right), \quad \textbf{(6)}$$

```
Input: Matrix[1..row,1..col]
Output: Matrix[1..row,1..col]

/*Matrix is a two-dimensional bit matrix containing
    the dependency graph*/
/*row is the number of row elements in the matrix
    corresponding to |T| + |C|*/
/*col is the number of column elements in the matrix
    corresponding to |T|*/

procedure CLOSEM (Matrix, row, col)
begin
    for k := 1 to col do
        for i := 1 to col do
            if ( (i<>k) & Matrix[i,k] = 1)) then
                for j := 1 to col do
                    Matrix[i,j] := Matrix[i,j] ∨ Matrix[k,j];
                endfor;
            endif;
        endfor;
    endfor;
    for i := row downto col + 1 do
        for j := col downto 1 do
            if (Matrix[i,j]) = 1) then
                for k := 1 to col do
                    Matrix[i,k] := Matrix[i,k] ∨ Matrix[j,k];
                endfor;
            endif;
        endfor;
    endfor;
end;
```

Figure 3. *Algorithm for computing transitive closure.*

$$\bar{\mathbf{D}}'_j = \left\{\hat{\mathbf{D}}'_{ij} \middle| \forall j\right\} \quad \left(\text{the } j\text{th-row vector of } \hat{\mathbf{D}}'\right), \tag{7}$$

and Fb_j denotes the feedback loop designated by the index of the first member of the group. Thus,

$$\mathbf{G}_i^{Fb_j} = \begin{cases} j; \left(\hat{\mathbf{D}}'_{ii} = 1\right) \wedge \\ \left(\bar{\mathbf{D}}'_i \equiv \bar{\mathbf{D}}'_j; \min_j \{0 < j \le i\}\right); \\ i \le |\mathbf{T}| \\ 0; \text{ otherwise} \end{cases} \tag{8}$$

where **T** is the set of all tests and |**T**| is the cardinality of that set.

A value of 0 indicates the test is not a member of any topological feedback loop. Every test with a value of j belongs to the same feedback loop. Many values of j may not be assigned. We can then determine conclusion participation in feedback by

$$\mathbf{G}_i^{Fb_j} = \begin{cases} j; \left(\hat{\mathbf{D}}'_{jj} = 1\right) \wedge \\ \left(\bar{\mathbf{D}}'_i \equiv \bar{\mathbf{D}}'_j; \min_j \{0 < j \le |\mathbf{T}|\}\right); \\ |\mathbf{T}| < i \le |\mathbf{T}| + |\mathbf{C}| \\ 0; \text{ otherwise} \end{cases} \tag{9}$$

Here, **C** is the set of all conclusions and |**C**| is the cardinality, and the value of j is the same as for Equations 6 and 7.

Conditional cross-checking. We define a conditional test to be a test for which the list of dependencies is conditioned by some state or mode of the system. Our implementation of the conditional test is currently limited to a set of mutually exclusive conditional states. Thus, conditional tests may depend only on nonconditional tests, conclusions, inputs, default conditional tests, and conditional tests of the same type. We define a *default conditional test* for all tests with conditionals to be the conditional test utilization in the absence of explicitly specified conditional information.

Given two tests t_x and t_y, and given conditionals A and B such that A is associated with t_x and B is associated with t_y, suppose that t_y conditioned on B depends on t_x conditioned on A. When the knowledge base is compiled, the following rules apply:

- *Rule 1.* If conditional A is the same as conditional B, then the dependency is acceptable.
- *Rule 2.* If conditional A is not the same as conditional B, and conditional B is not the default conditional for t_y, then the dependency is not acceptable because a nondefault conditional test may depend only on another test of the same condition or on a default conditional test.
- *Rule 3.* If conditional A is not the same as conditional B, and conditional B is both the default conditional for t_y and the default conditional for t_x, then the dependency is not acceptable because the mutually exclusive assumption would limit t_y conditioned on B to depend on t_x conditioned on B.
- *Rule 4.* If conditional A is not the same as conditional B, and conditional B is the default conditional for t_y, then the dependency is acceptable.

The system considers violations of Rules 2 and 3 as cross-conditionals. The analyst can evaluate some of these rules while creating the model. However, transitive closure may cause violations of these rules

to appear that could not readily be identified during input. In addition, logical closure may result with the occurrence of cross-conditionals. However, these cross-conditionals are logical artifacts of the model and need not be eliminated. As a result, the software performs conditional cross-checking between transitive and logical closure.

Logical closure. The next step in compiling the knowledge base is to examine the matrix for additional inferences that may be drawn. These additional inferences are based on a closed-world assumption expressed for diagnosis as follows:

- If a test with an unknown outcome depends on all of the unknown elements in the model, then that test must have a failed outcome.
- If a test with an unknown outcome depends only on elements known to have passed, then that test must have a passed outcome.

The closed-world assumption is permissible because of the existence of the special primitive element called *No Fault*, upon which no test depends. This element prevents a test from being declared bad under these inference rules as long as the *No Fault* conclusion is still a consideration.

We refer to the identification of dependencies through these rules as logical closure. Mathematically, logical closure can be represented as follows: Given two tests, t_x and t_y, let $\mathbf{A}^*$ be the set of conclusions on which t_x depends (c_a and c_b) and $\mathbf{B}^*$ be the set of conclusions on which t_y depends (c_a). If $\mathbf{B}^*$ is a subset of $\mathbf{A}^*$, then we can infer that t_x depends on t_y. For example, suppose we have the following dependencies in our model:

- t_x depends on c_a
- t_x depends on c_b
- t_y depends on c

If we assume that all other dependencies for t_x and t_y are the same, we can conclude that t_x depends on t_y. Thus, logical closure consists of examining the test-to-conclusion dependencies following transitive closure. When a subset relationship exists as described here, the algorithm inserts a new dependency into the test-to-test portion of the dependency matrix. The process of logical closure, shown in Figure 4, is tied to incremental closure.

Incremental closure. When we add a dependency through logical closure, new higher order dependencies may exist. These new dependencies do not affect the test-to-conclusion portion of the matrix, so we need to "close" the new bit only in the test-to-test portion of the matrix. (In the event a new bit is added in the test-to-conclusion dependencies, the algorithm can incrementally close this new bit into the model by applying logical closure on the corresponding column.) This process, called incremental closure, is shown in Figure 5. Once the matrix is closed, it is ready for the remaining analyses. If we do not know the test-to-test relationships but do know the higher order test-to-conclusion relationships, then logical closure will bring the full relationship map back. We call this type of model development *attribute mapping*.[1]

Processing input data

Figure 6 on page 34 shows the results of applying the five major steps in compiling the knowledge base.

- *Transitive closure.* Bits added to the matrix as the result of transitive closure are indicated by an "h."

```
Input: Matrix[1..row,1..col]
Output: Matrix[1..row,1..col]

/*Matrix is a two-dimensional bit matrix containing
   the dependency graph that has*/
/*already been through the transitive closure algorithm*/
/*row is the number of row elements in the matrix
   corresponding to |T| + |C|*/
/*col is the number of column elements in the matrix
   corresponding to |T|*/

procedure LOG_CLOSE (Matrix, row, col)
begin
   for k := 1 to col do
      for j := 1 to col do
         if (i <> j) then
            flag := 0;
            for i := row downto col +1 do
               if (Matrix [k, i] <> Matrix [j,i]) then
                  if (Matrix [k,i] = 0) then
                     flag := 1; /*no relationship exists*/
                  endif;
               endif;
            endfor;
      endfor;
end;
```

Figure 4. *Algorithm for computing logical closure.*

- *Feedback analysis.* Feedback is indicated whenever an "h" appears in a diagonal cell of the higher order dependency matrix. (We will address the feedback analysis of the case study in the next article in this series.)
- *Conditional cross-checking.* The case study contains no conditional tests.
- *Logical closure.* Bits added to the matrix as the result of logical closure are indicated by an "l."
- *Incremental closure.* Bits added to the matrix as the result of incremental closure are indicated by an "n."

Capitalizing on logical closure

The process of creating models can be complicated and time-consuming. Therefore, we concentrated much of our research on approaches to simplifying the modeling task. We found that the algorithm for logical closure provides an excellent opportunity for such simplification. The literature describes two analyses that can assist the modeling process

- the failure mode, effects, and criticality analysis, or FMECA,[12] and
- the fault dictionary.[13]

```
Input: Matrix[1..row,1..col], new_bit_row,
  new_bit_col, row, col
Output: Matrix[1..row,1..col]

/*Matrix is a two-dimensional bit matrix containing
  the dependency graph that has*/
/*already been through the transitive closure algorithm*/
/*row is the number of row elements in the matrix
   correspondingto | T | + |C|*/
/*col is the number of column elements in the matrix
  corresponding to | T |*/
/*new_bit_row is the row of a logically created bit*/
/*new_bit_col is the column of a logically created bit*/

procedure INC_CLOSE(new_bit_row, new_bit_col,
  Matrix, row, col)
begin
  for k := 1 to col do
     Matrix[k,new_bit_col] := Matrix[k,new_bit_col]
          ∨ Matrix[k,new_bit_row];
  endfor;
  for k := 1 to col do
     if (Matrix[new_bit_col,k] = 1) then
        for j := 1 to col do
           Matrix[j,k] := Matrix[j,k] ∨
              Matrix[j,new_bit_col];
        endfor;
     endif;
  endfor;
end;
```

Figure 5. Algorithm for computing incremental closure.

A FMECA describes the symptoms associated with each failure mode of a system and estimates the impact of the failure mode on mission success, safety, system performance, maintainability, and maintenance requirements. A fault dictionary is a table, indexed by faults, containing lists of failures that may have caused a listed symptom.

We call the symptoms associated with a fault the attributes of the fault. If we associate a failed test (or a set of failed tests) with each symptom, when a fault may have caused a particular symptom to appear, the test associated with the symptom depends on the fault. (In part 1,[1] we say that a test depends on tests and conclusions that may cause that test to fail.) Because the test associated with the symptom depends on the fault, we can use the FMECA and the fault dictionary to determine the set of dependencies between the tests and the failure modes of the system the analyst is modeling. The complete set of conclusion-to-test attributes is called the attribute map, and the complete set of test-to-conclusion dependencies for a system is a consequence of the attribute map. Given an attribute map of the system, logical closure can determine test-to-test dependencies, thus completing the higher order model.

One approach to generating the attribute map involves creating a simulation model of the system to be analyzed. We can develop the simulation model using any of the standard simulation tools (for example, PSPICE). Once the model is complete and running, we can enter tests into the model as observation points or probe points. (We should define tests so they can be executed without reliance on previous states. We call a test defined in this way an *encapsulated test.* We will address encapsulated tests in a future article.) Then we construct the attribute model in the following steps:

1. Run the simulation model without failures and record the measurements at each of the simulated test points. The measurements represent

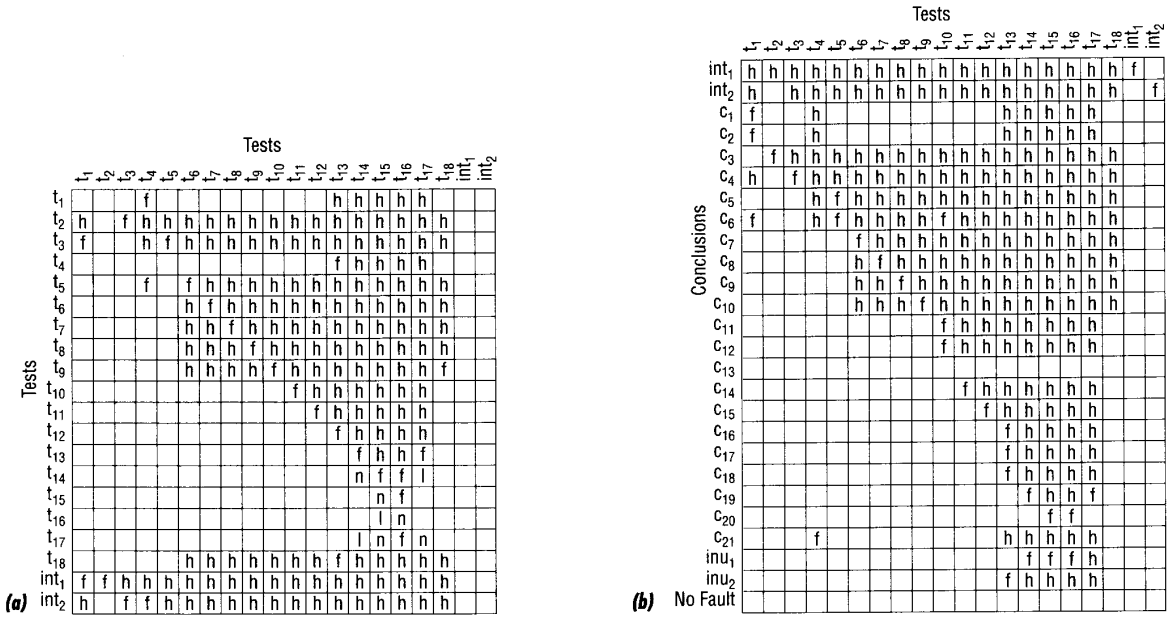

Figure 6. *Closed dependency matrices for the case study: Test-to-test dependency matrix (a) and test-to-conclusion dependency matrix (b). h = a bit resulting from transitive closure, l = a bit resulting from logical closure, and n = a bit resulting from incremental closure.*

nominal readings for each test.

2. Specify a nominal range for each of the tests, using the simulation of the nominal case.
3. Independently insert a failure for each component into the simulation model according to each appropriate component failure mode.
4. Run the simulations for each of the failure modes and record the measurements at each of the simulated test points. The measurements represent the attributes of the particular failure mode.
5. If the values of a test measurement are beyond the corresponding nominal range, enter the failure mode as a dependency of that test.

Once we have completed all these steps, the attribute map determines the complete list of test-to-conclusion dependencies. We complete the model by performing logical closure on the attribute map.

Modeling the case study

For the system shown in Figure 1, we can obtain an attribute map by using a FMECA, a fault dictionary, or a simulation of the system, or by exhaustively tracing the dependency chart. (Some levels of learning may be able to recover flaws in this mapping process. We will discuss machine learning as it applies to this problem in a future article in this series.) For example, if c_{21} fails, then t_4, t_{13}, t_{14}, t_{15}, t_{16}, and t_{17} will detect the failure.

Figure 7 on the next page shows the dependency matrix for the complete attribute map for this system. Bits derived by attribute mapping are indicated by an "m." Under this model-building approach, transitive closure is not appropriate because no test-to-test dependencies exist. As a result, topological feedback analysis and conditional cross-checking analysis are also inappropriate.

Figure 8 shows the logically closed matrix of Figure 7. This closed matrix contains the same bit representation shown in Figure 6. The major difference is where the bits come from. In Figure 8, logical bits, as derived from the algorithm in Figure 4, are then incrementally closed using the procedure shown in Figure 5. Which bits have an "l" and which have an "n" depends on the order of calculation. For example, numbering the tests or components differently affects the source of the higher order bits but not the final higher order representation.

Test paradigms

Several test paradigms may be used in modeling a system such as the antitank missile launcher. In addition to symmetric test paradigms, combinations of test paradigms are also possible, such as special, conditional, cross-linked, and asymmetric.

Symmetric test paradigm. To this point we have assumed that the tests in

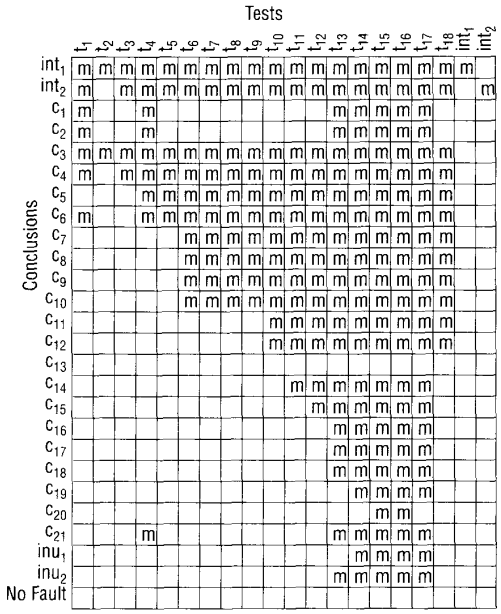

Figure 7. *Attribute-mapped dependency matrix for the case study. Only the test-to-conclusion dependency matrix is shown because the test-to-test dependency matrix is empty. m = a bit derived by attribute mapping.*

our model are symmetric. We define a symmetric test as a test that provides complementary information given a pass outcome and a fail outcome.[1] Symmetry can be shown graphically, as in Figure 9 on the next page. Note that the elements that can be determined to be good following a passed test (Figure 9a) are the same elements still under consideration following a failed test (Figure 9b). The bits entered into the dependency matrix are shown in Figure 2. In general, tests may not always be symmetric.

Special test paradigm. The special test is a subset of other test paradigms and includes only fault-isolation conclusions in its dependency list. The term *special test* applies to the familiar form of special testing where a test is devised to examine a specific function or piece of hardware. Although a special test is set up without test-to-test dependencies, logical closure may create these dependencies in the higher order representation. Under the

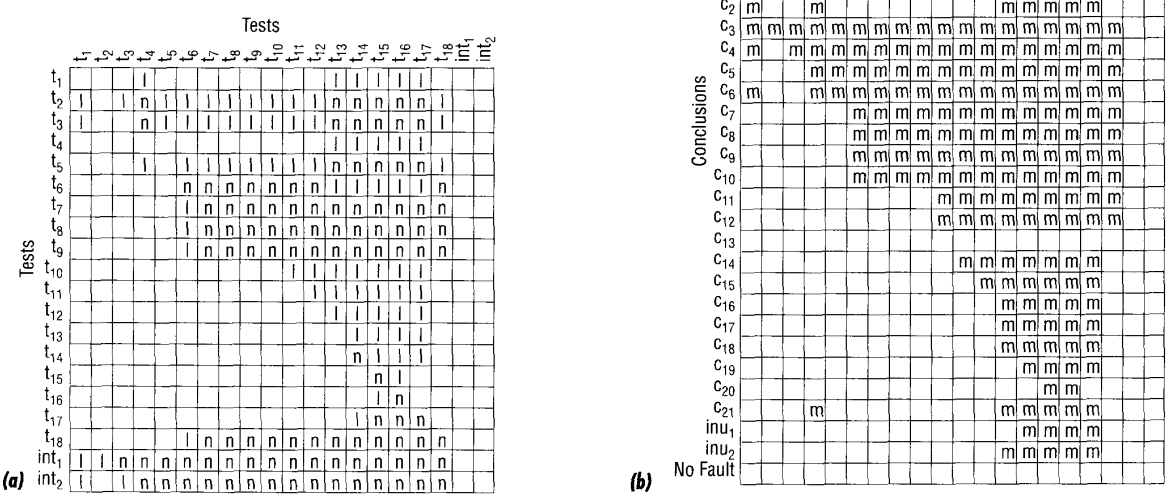

Figure 8. *Logically closed attribute-mapped dependency matrix for the case study: Test-to-test dependency matrix (a) and test-to-conclusion dependency matrix (b).*

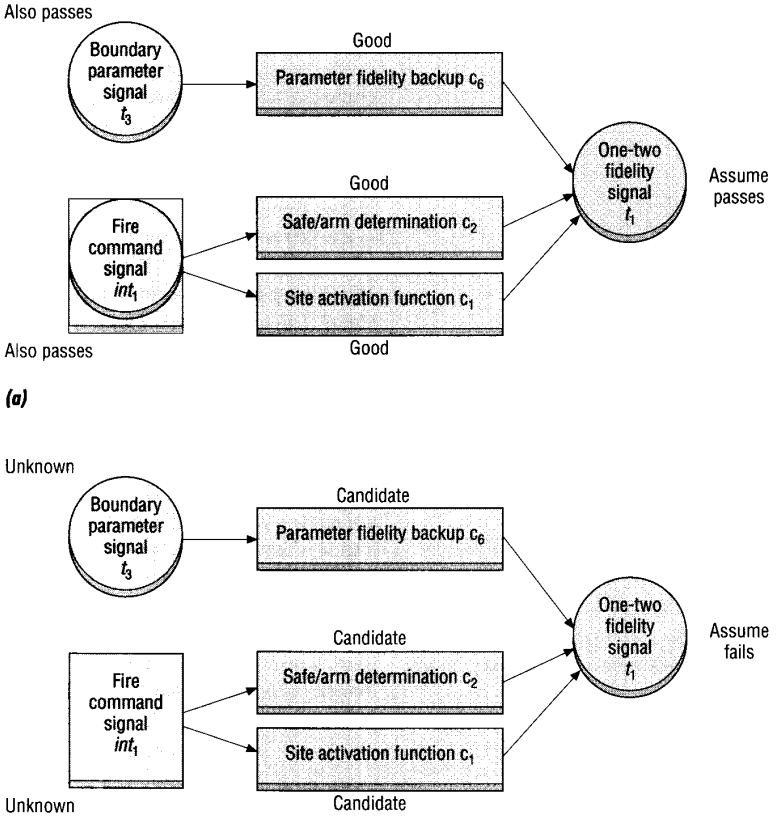

Figure 9. *Dependency graph for a symmetric test: Information gained from a passed test (a) and information gained from a failed test (b). In (b) downstream tests will fail, and downstream components will be masked.*

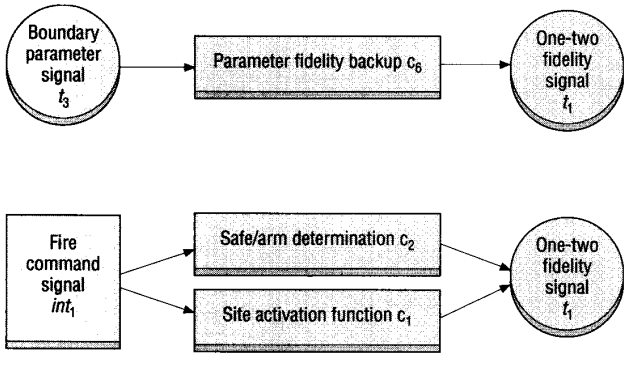

Figure 10. *Dependency graph for a conditional test: Polarized filter off sight (a) and polarized filter on sight (b).*

matrix formulation, every test in an attribute-mapped model can be said to be a special test. Special tests may, in general, have the properties of any of the other test paradigms.

Conditional test paradigm. One alternative to the symmetric test paradigm is the conditional test paradigm. Figure 10 shows an example dependency graph. The example test has two conditions represented as two mutually exclusive dependency graphs. In representing a conditional test in the matrix, we create one test image for each condition. For the case study, we would have two columns and two rows for test "One-two fidelity signal," each having different dependencies. A conditional test may have the properties of any of the other test paradigms.

Cross-linked test paradigm. The basic inference mechanisms associated with the standard symmetric test limit inference to the same truth value as the test outcome. For example, if t_1 passes, we can conclude that all tests that feed t_1 in the higher order dependency matrix (Figure 6 or Figure 8) pass (that is, t_2, t_3, int_1, and int_2). On the other hand, if t_1 fails, then we can conclude that all the tests that t_1 feeds will also fail (that is, t_4, t_{13}, t_{14}, t_{15}, t_{16}, and t_{17}).

The basic dependency formulation lacks a mechanism by which we can infer cross-linked outcomes. For example, if t_1 passes, we may infer that t_{11} fails. Cross-linkages of this type may be represented outside the matrix as a trigger for the inference engine and the test choice process to be discussed in later articles. Figure 11 illustrates the dependency representation. The two tests in the figure are represented as normal symmetric tests in the dependency matrix, and the cross-linkage (shown as a dashed line) is stored outside the matrix. Cross-linked tests may, in general, have the properties of any of the other test paradigms.

Asymmetric test paradigm. The asymmetric test paradigm considers tests in

which the inferences drawn from an outcome are not complementary. For example, Figure 12 shows the dependency graph for a test where a pass outcome (Figure 12a) results in a list of inferences as shown, but a fail outcome (Figure 12b) provides no information at all. It is possible for the asymmetric test to provide an alternative, noncomplementary set of inferences rather than suppressing all inferences for one of the outcomes. We represent an asymmetric test as two test images in the dependency matrix—one image for inferences drawn when the test passes, and one for inferences drawn when the test fails. These two images are directly linked, and the direct linkage is stored outside the matrix.

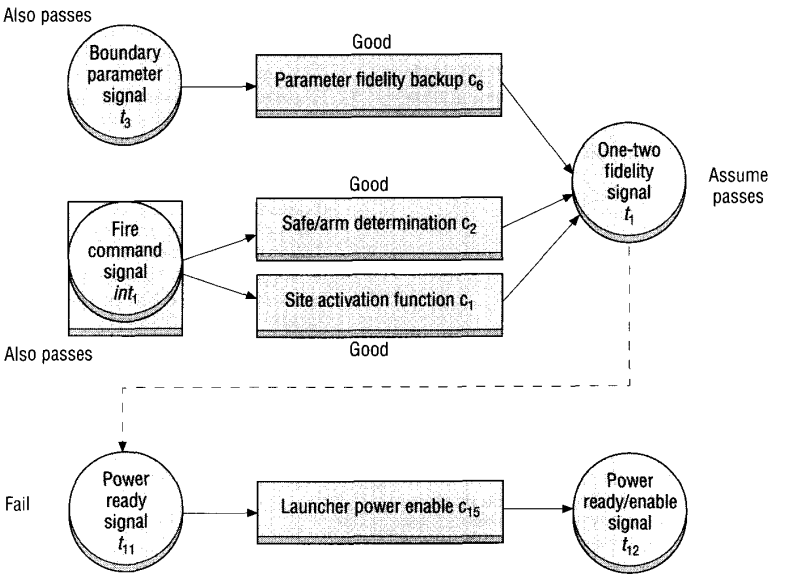

Figure 11. *Dependency graph for a cross-linked test.*

The modeling technique developed for the analysis of testability and the diagnosis of complex electronic systems has been used in a much wider framework. Simpson and Bailey[14] describe its use in a noncooperative identification problem. Here, information sources are sensor readings obtained from multiple sensor types, and conclusions are related to identification of the target aircraft type.

McNamara[15] describes the use of the modeling technique in a layered reservation-access token-passing scheme. Sheppard[16] describes the modeling approach as it applies to knowledge-base verification and validation. The modeling approach has also been used in electronic warfare signal sorting,[17] medical diagnosis,[18] and avionics configuration control.[19]

An information flow model for integrated diagnostics includes topological, logical, and set membership features. Compiling such a model provides all the information normally found in backward and forward chaining as computed in rule-based systems. The information flow model allows us to conduct a number of analyses of the testability of the system and to explore various diagnostic techniques for constructing fault trees and providing an interactive diagnosis capability.

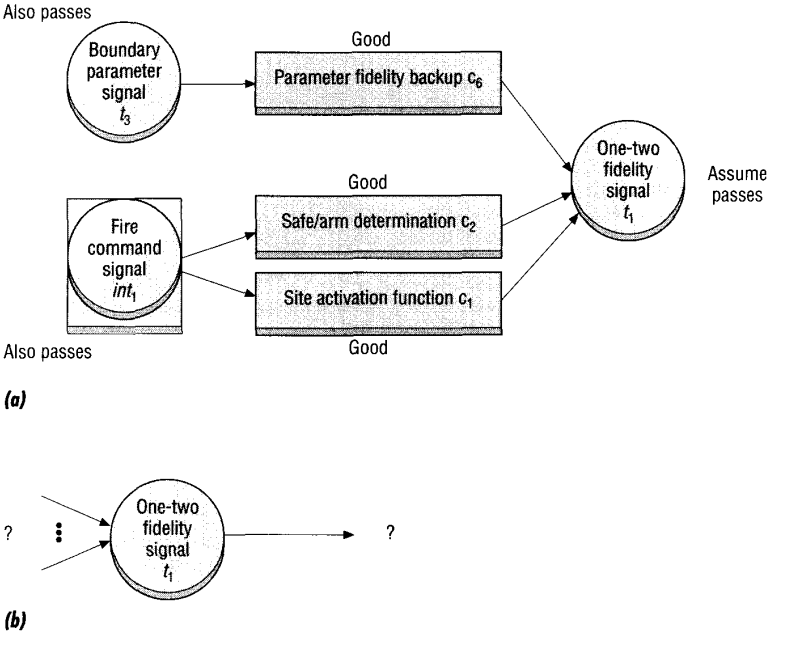

Figure 12. *Dependency graph for an asymmetric test: Information gained from t_1 passing (a) and information gained from t_1 failing (b).*

Subsequent articles will derive a framework for testability analyses using this model and provide algorithms for choosing tests for fault trees, intelligent maintenance aids, automatic test equipment, or embedded diagnostics. D&T

References

1. W. Simpson and J. Sheppard, "System Complexity and Integrated Diagnostics," *IEEE Design & Test of Computers*, Vol. 8, No. 3, Sept. 1991, pp. 16-30.
2. M. Labit et al., "Special Report on Operational Suitability (OS) Verification Study Focus on Maintainability," tech. report. 1751-01-02-2395, Arinc Research Corp., Annapolis, Md., 1981.
3. "Avionics Maintenance Conf. (AMC) Report-San Diego, 1987," tech. report 87-087/MOF-34, Aeronautical Radio, Inc., Annapolis, Md., 1987.
4. R. DePaul, Jr., "Logic Modeling as a Tool for Testability," *Autotestcon 85 Conf. Record*, IEEE Press, New York, 1985, pp. 203-207.
5. R. Cantone and P. Caserta, "Evaluating the Economical Impact of Expert Fault Diagnosis Systems: The I-CAT Experience," *Proc. Third IEEE Int'l Symp. Intelligent Control*, IEEE Computer Society Press, Los Alamitos, Calif., 1988.
6. J. Franco, "Experiences Gained Using the Navy's IDSS Weapon System Testability Analyzer," *Autotestcon 88 Conf. Record*, IEEE Press, New York, 1988, pp. 129-132.
7. K. Pattipati, "START: System Testability and Research Tool," *Autotestcon 90 Conf. Record*, IEEE Press, New York, 1990, pp. 395-402.
8. *Testability Program for Electronic Systems and Equipment, MIL-STD-2165*, Naval Electronics Systems Command (ELEX-8111), Washington, DC, 1985.
9. F. Johnson and R. Unkle, "The System Testability and Maintenance Program (STAMP): A Testability Assessment Tool for Aerospace Systems," *Proc. AIAA/NASA Symp. on the Maintainability of Aerospace Systems*, AIAA, New York, 1989.
10. J. Sheppard and W. Simpson, "Incorporating Model-Based Reasoning in Interactive Maintenance Aids," *Proc. 42nd National Aerospace and Electronics Conf.*, IEEE Press, New York, 1990, pp. 1238-1242.
11. A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974, pp. 199-206.
12. *Procedures for Performing a Failure Mode, Effects, and Criticality Analysis, MIL-STD-1629A*, Naval Aviation Systems Command (NAVAIR-5112), Washington, DC, 1988.
13. J. Pennell, L. Haynes, and J. Salasin, "A Proposed Architecture for Integrated Diagnosis," *Autotestcon 88 Conf. Record*, IEEE Press, New York, 1988.
14. W. Simpson and J. Bailey, "The ARINC Noncooperative All-Source Target Identification (NASTI) Program," *Tri-Service Combat Identification System Conf.*, Naval Air Develoment Center, Warminster, Pa., 1984.
15. B. McNamara, "AI Concepts in Mux Bus Control: Layered Reservation Access," *IEEE/AIAA 6th Digital Avionics Systems Conf.*, IEEE Press, New York, 1984.
16. J. Sheppard, "An Approach to Verifying Expert System Rule Bases," presented at *IEEE Int'l Conf. Systems, Man, and Cybernetics*, 1989; contact J. Sheppard, Arinc Research Corp., Annapolis, Md.
17. W. Simpson and B. McNamara, "The ARINC Research Signal Evaluation for Emitter Recognition (SEER) Program," *Joint Western-Mountain Region Tech. Symp.*, Assoc. of Old Crows, Warner Robins, Ga., 1985.
18. W. Simpson et al., "An Artificial Intelligence Approach to Developing Medical Protocols," *Proc. Fifth World Congress Medical Informatics*, Elsevier Science Publishers, Amsterdam, 1986, pp. 776–780.
19. S. Baily and B. Kelley, "Aeronautical Systems Workbench: A Concept," *Eighth Ann. IEEE/AESS Dayton Chapter Symp. Avionics in Conceptual System Planning*, IEEE CS Press, 1986.

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, an intelligent, interactive maintenance aid, 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 Johns Hopkins University—both in computer science.

William 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. 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 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.