Applying Testability Analysis for Integrated Diagnostics

JOHN W. SHEPPARD
WILLIAM R. SIMPSON
Arinc Research Corporation

In part 4 of their series, the authors discuss a key element in the integrated diagnostic process: applying the testability measures described in part 3 to improve system design and thus improve system testability. The authors also explore alternative approaches to improving system testability and describe additional applications of testability analysis.

AN IMPORTANT STEP in the integrated diagnostic process is developing requirements and gathering resources to test a system. In the third article in this series,[1] we described a number of measures that evaluate system testability at a specific time during system design. In this article, we demonstrate how to use these measures to improve testability as part of an iterative design process. Our objective is to minimize ambiguity in replaceable unit groups (defined as a group of fault isolation conclusions treated as one isolatable element), while also minimizing the amount of testing and the number of tests.

We assume that the system is designed to perform a specific job and that the next iteration in the design process will improve system testability from a maintenance perspective. Any changes made to improve testability should not alter system performance during normal operational modes.

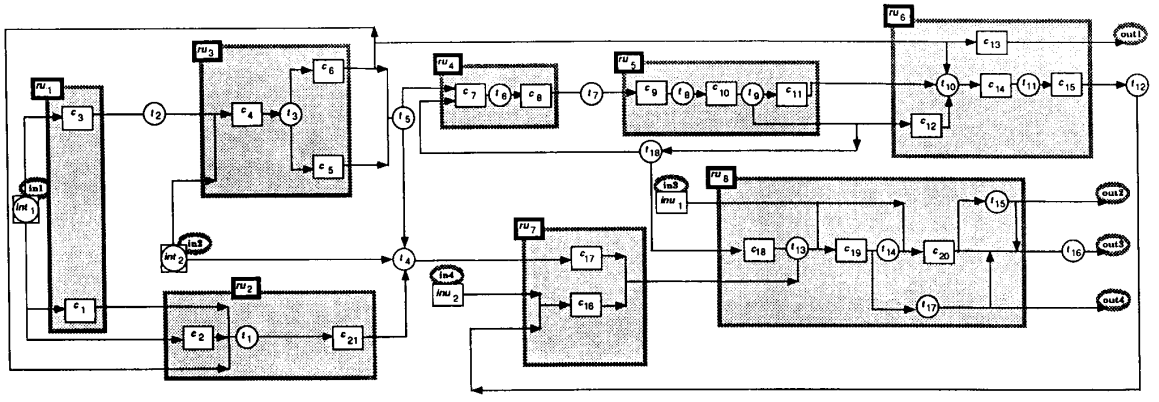
Throughout our discussion, we refer to the case study of an antitank missile launcher introduced in our second article.[2] Figure 1 is a dependency diagram for the case study. Tables 1 and 2 list the tests and conclusions in the case study.

Motivation for improving testability

In recent years, the idea of system testability has changed from a mere afterthought to an integral part of the design process. Designers now emphasize field maintainability early in the life cycle, to maximize downstream ability to test the system and diagnose faults, and to minimize the likelihood of retest OK (RTOK) and cannot duplicate (CND) events.[3,4] Recall that a RTOK event occurs when a unit thought to be not functioning nominally at one level of maintenance is replaced and then found to be functioning nominally at the next level. A CND event occurs when a fault indication is not repeatable at the same level of maintenance. We perform diagnostic testing for two basic reasons: to verify that the system is operational and behaves nominally, and to localize a fault to a degree consistent with the level of repair.

In addition to improving testability, we want to provide a method of performing testability analysis and diagnosis in an integrated diagnostic framework.[5,6] This framework requires a structured process in which diagnostic data are available at all maintenance

0740-7475/92/0900-0065$03.00 © 1992 IEEE



Legend:				
=Conclusion	=Test	=Testable input	=Replaceable unit group	=System input or output
21 conclusions + 4 inputs + No fault = 26	18	2	8 replaceable unit groups + 4 inputs + No fault = 13	8

Figure 1. *Dependency diagram of antitank missile launcher case study.*

Table 1. *Tests in the case study.*

Available tests	Label*
Fire command signal	int_1
Reticle position tracker	int_2
One-two fidelity signal	t_1
Readiness output signal	t_2
Boundary parameter signal	t_3
Summary safe/arm signal	t_4
Combined fidelity signal	t_5
Error signal	t_6
Course correction signal	t_7
Command corrector signal	t_8
Command response signal	t_9
Track signal	t_{10}
Power ready signal	t_{11}
Power ready/enable signal	t_{12}
Bounded error signal	t_{13}
Launcher enable signal	t_{14}
Launch track generator signal	t_{15}
Launch track evaluation signal	t_{16}
Launcher ready signal	t_{17}
Stabilized loop signal output	t_{18}

* int *corresponds to a testable input, including both the test and conclusion element, and* t *corresponds to a test.*

levels and complete system diagnosis is accomplished in a centralized, hierarchical manner. In the second article of this series, we proposed a technology-independent, hierarchical mathematical model of such a framework.

Review of the model

We are interested in measuring the flow of diagnostic information through a system. This measuring process is primarily inference-based; that is, for each test in the model, we ask what we learn from a pass or a fail test outcome. In characterizing the model, we described several test variants that have the effect of modifying the inference process. For example, the asymmetric test allows inference to be suppressed for either a pass or a fail outcome.

The two primitive elements in the information flow model are the test and the fault isolation conclusion. Because we are interested in the logic of the diagnostic system, tests are not required to take on any particular form; rather, tests are simply information carriers. Thus, we allow a test to be a stimulus-response pair, a BIT (built-in test) indication, or a simple observation (such as a scorched area around a pin). Again, because of the logical basis, fault isolation conclusions correspond to the set of possible conclusions that we can draw using the current set of tests. Therefore, in a single model, conclusions may be failures in functionality, the identification of a box, an electronic component, or a component failure mode. Note that our definition does not limit a conclusion to an implemented function. Nor does it limit a conclusion to a failure mode. The model leaves the mapping between the conclusion and a system element at the disposal of the modeler.

One advantage of the information flow model is that we can readily represent it in a form that facilitates analysis—in particular, an adjacency matrix corresponding to a graph of the model's topology. Figure 2 (on page 68) is an adjacency matrix representation of the case study. Each occupied matrix cell represents a relationship between two elements in the model, and the value in the cell indicates how that relationship was developed. When we represent the model in matrix form, we can manipu-

late the matrix to determine testability characteristics and to develop efficient fault isolation strategies.

Review of the analyses

The information flow model allows us to identify several characteristics of system testability, such as conclusion ambiguity and test redundancy. Conclusion ambiguity occurs when the current set of tests cannot distinguish among two or more conclusions. This situation is distinct from the desire to draw two or more conclusions, which is called the multiple-conclusion problem (to be addressed in a future article). Test redundancy occurs when two or more tests provide identical logical information about the system.

A redundant test is a form of excess test. A test is considered excess when some combination of other tests provides the same information it does. We are interested in identifying excess tests because they allow us to make engineering trade-offs to obtain the best combination of tests for the stated testing objectives.

Tools and methods for analyzing system testability often incorporate the assumption that only one failure will occur at any time. This assumption permits us to greatly reduce the search space.[7] Nevertheless, system designers need to know the problems that may result from the presence of multiple failures. The information flow model enables the analyst to predict two difficult multiple-failure problems: the root cause failure and the false failure. Any fault may hide an additional failure from detection. A root cause failure occurs when one of these hidden failures is the cause of an isolated fault. Repair of the isolated fault alone is not effective because the root cause will make the repaired unit fail again. In addition, it is possible for two or more faults to produce symptoms identical to some other fault in the system. In that case, repair of the isolated fault is ineffective because the fault was not truly a fault in the first place.

Table 2. *Conclusions in the case study.*

Fault isolation conclusion	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	ru_1
Safe/arm determination	c_2	5	ru_2
Launcher ready evaluator	c_3	100	ru_1
Boundary check	c_4	100	ru_3
Parameter fidelity	c_5	100	ru_3
Parameter fidelity backup	c_6	100	ru_3
Error evaluator	c_7	5	ru_4
Error corrector	c_8	5	ru_4
Command signal evaluator	c_9	5	ru_5
Response generator	c_{10}	5	ru_5
Target tracker	c_{11}	5	ru_5
Command to track comparator	c_{12}	5	ru_6
Guidance output	c_{13}	5	ru_6
Launcher power supply (battery)	c_{14}	100	ru_6
Launcher power enable	c_{15}	100	ru_6
Launcher power signal conditioner	c_{16}	5	ru_7
Cross-check override	c_{17}	5	ru_7
Error signal boundary check	c_{18}	5	ru_8
Launcher full ready function	c_{19}	5	ru_8
Launch command function	c_{20}	100	ru_8
Fire ready activation function	c_{21}	100	ru_2
No fault found	*No Fault*	9,095	0

* 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.*

† *System inputs are not listed as members of replaceable unit groups, although they could be.*

Our analyses permit the engineer to assess both a system's inherent testability (that is, the maximally achievable testability) and its achieved testability (testability resulting from the defined set of tests). The information flow model follows the flow of diagnostic information through the system and represents the logical relationships between tests and conclusions used in diagnosis. The model's composition is intimately tied to information provided by tests. Thus, modeling system connectivity provides information on inherent testability; modeling information provided by actual tests yields an analysis of achieved testability.

Interpreting the testability measures

In our last article we discussed measures that evaluate the testability of a system. These measures are derived from the information flow model and therefore are actually measures of the model. Specifically, we developed measures associated with ambiguity, feedback, the test set, and multiple failures. We in-

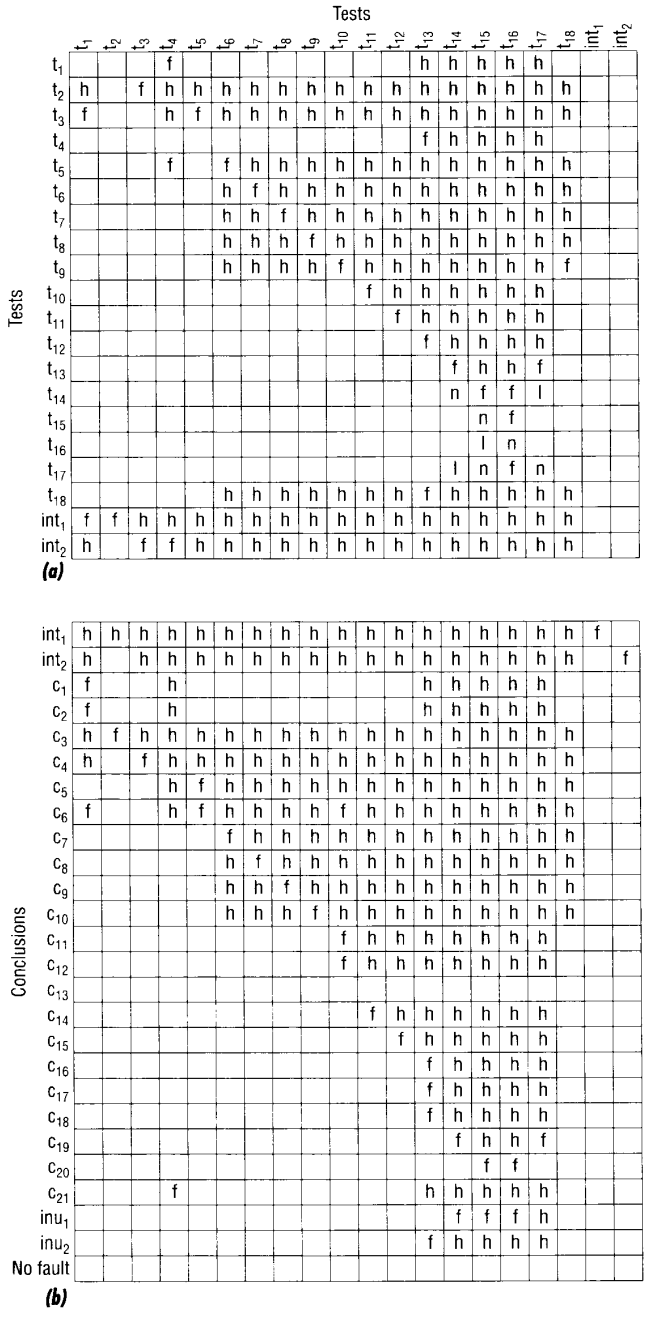


Figure 2. *Closed dependency matrices of case study system: test-to-test dependency matrix* ***(a)****; test-to-conclusion dependency matrix* ***(b)****. f = a first-order dependency; h = a bit resulting from transitive closure; l = a bit resulting from logical closure; n = a bit resulting from incremental closure.*

tended the measures to indicate system testability, not to set firm thresholds or standards by which to determine if a system has good or bad testability. Rather, one must interpret the measures in the context of a specific system and its set of requirements.

In the earlier article we presented algorithms for computing the testability measures, and we gave ideal values and values for the case study. Now we will review three of the measures—operational isolation, test redundancy, and excess tests—so that we can make decisions about the testability of the case study and formulate recommendations for improving system testability.

Operational Isolation. Operational isolation (OI[*n*]) indicates potential RTOK problems due to high ambiguity in a fielded system. Operational isolation is the percentage of time the test set will fault-isolate to *n* or fewer replaceable unit groups.

We compute operational-isolation variations based on information about the system and on the subset of conclusions to be considered. We will consider four such variations. The first, OI_u, assumes a uniform failure rate for all the individual components that make up the replaceable unit groups and considers all fault isolation conclusions. The second, OI_w, assumes individual failure rates for all the components and also considers all fault isolation conclusions. OI_w corresponds to an *expected* percentage of the time that fault isolation will be at *n* or fewer replaceable unit groups. The third and fourth variations are both weighted, but one (OI_{in}) ignores system inputs, and the other (OI_{nf}) ignores system inputs in combination with *No Fault*. These two variations give us the options of considering the system independent of incoming information and when a failure has been detected.

Figure 3 presents operational isolation values for the case study. We can immediately see that the system has a

problem with ambiguity. Because none of the operational isolation measures indicate the ability to fault-isolate to single replaceable unit groups 100% of the time, the system clearly exhibits ambiguity between at least two replaceable unit groups. In fact, for both the weighted and the unweighted OIs, ambiguity exists between three replaceable unit groups. Note, however, that excluding inputs reduces ambiguity to two replaceable unit groups at most. Therefore, a major contributor to ambiguity is our inability to directly observe the system inputs.

When we compare OI_{in} and OI_{nf} for one replaceable unit group, we see that a second contributor to ambiguity is our inability to detect some failure. Further analysis of the system reveals that we cannot detect the failure of c_{13} with the current set of tests. This, in fact, is the only nondetection. Because c_{13} is not detected, it will be ambiguous with *No Fault*. Furthermore, the extreme difference between the two values is a result of the fact that the failure rate of *No Fault* (that is, the probability that *No Fault* will be concluded) is extremely high—0.9095 (Table 2). Thus, we have determined that additional testability is necessary to detect a previously undetectable failure.

We used this analysis to evaluate the testability of the air pressurization system for an 11-MW fuel cell power plant.[8] The system-level analysis determined that only 33% of the system could be uniquely isolated during system start-up and 60% during its operational mode. We developed additional tests that increased operational isolation to 75% during start-up and 75% during operation.

Excess-test measures. Not only may a system have ambiguity problems (and therefore operational-isolation problems), but inappropriately placed tests may lead to overspecification of testability in particular areas of the system. The test redundancy (TR) and excess-test (XM) measures indicate overspecification. We can combine these measures with other measures to determine other types of problems in the system.

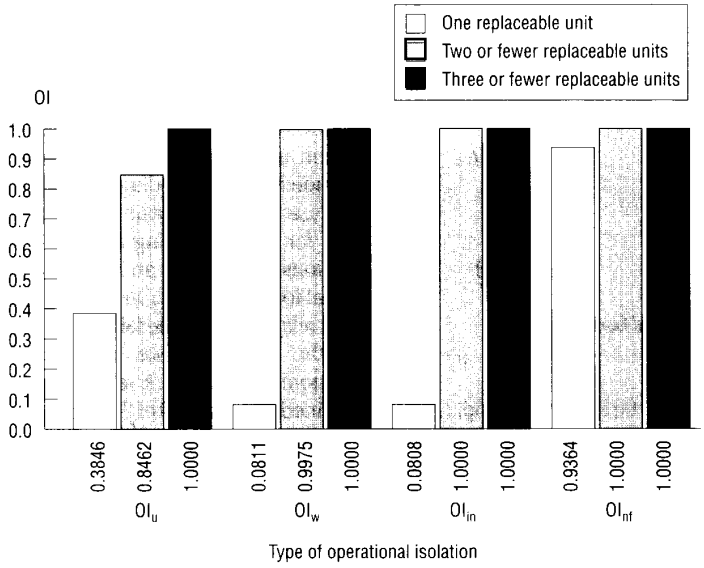


Figure 3. *Operational isolation (OI) in the case study.*

The test leverage (TL) measure provides a general indication of how well testability is specified for the system. For the case study, we found that TL = 0.7692, which lies between the bounds recommended by the theoretical minimum test leverage (0.1808) and the theoretical maximum test leverage (0.9615). The test leverage, however, is based on no ambiguity.

We determined that for the case study, TR = 0.3; that is, 30% of the tests in the model provide completely redundant information with some other test or tests. One way test redundancy arises is that the system contains information flow feedback. When we examine the feedback-modified test leverage, we find that FMTL = 0.6957, which is less than TL. Therefore, we know that the system also has a problem with feedback. Indeed, the value of test feedback dominance (TFD) for the case study is 0.25; that is, 25% of the information sources are tied up in feedback. Table 3 lists the ambiguity groups in the case study, including those involved in feedback. Determining a way to break the feedback loop should reduce the redundancy

Table 3. *Ambiguity groups in the case study.*

Group	Members
1	c_1, c_2
2*	c_7, c_8, c_9, c_{10}
3	c_{11}, c_{12}
4**	c_{13}, *No Fault*
5	c_{16}, c_{17}, c_{18}, inu_2
6	c_{19}, inu_1

* *Part of a topological feedback loop.*
** *Any ambiguity with* No Fault *is a nondetection.*

Table 4. Redundant- and excess-test groups.

Group	Tests
1*	t_6, t_7, t_8, t_9, t_{18}
2	t_{14}, t_{17}
3	t_{15}, t_{16}
4	t_5 (excess)
5	int_1 (excess)

*Topological feedback

provided by the tests in the loop. (Alternatively, we could design the feedback loop so that it is located on a single replaceable unit group, eliminating all but one test.)

In addition to determining TR, we found that the excess-test measure for conclusions including inputs (XMIC) is 0.556, and the excess-test measure for replaceable unit groups including inputs (XMIR) is 0.9. These tests provide the same information as some combination of other tests and are candidates for elimination. An excess-test analysis of the system reveals that t_5 and int_1 may be considered excess. Table 4 lists all the redundant and excess tests.

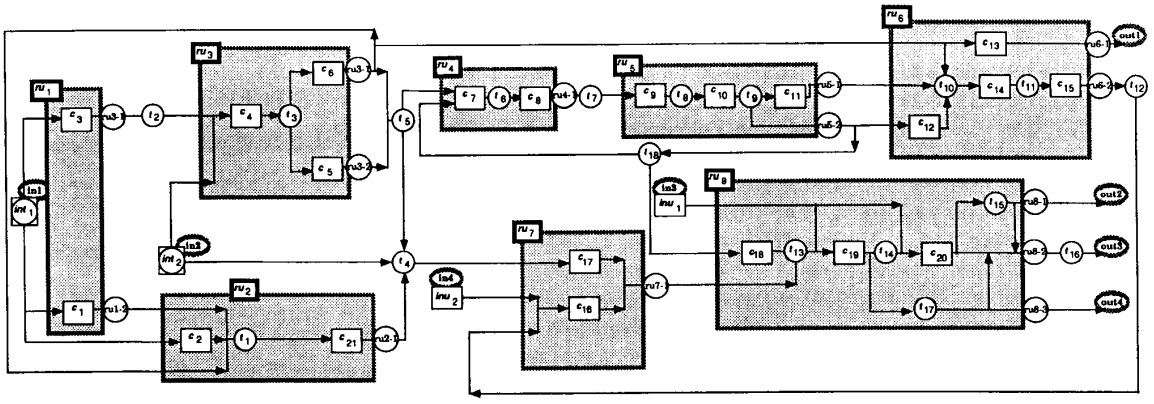
Removing all redundant and excess tests may have detrimental effects on overall system testability, depending on system requirements. Potential effects include less efficient diagnostic procedures, a decrease in our ability to use tests for cross-checking and verifying previous results (thus an increased potential for false alarms), and more problems related to multiple failures. In work for the US Army, we analyzed an electronic-warfare track vehicle and determined that only 650 of 2,000 designed tests were necessary for unambiguous fault isolation.[9] On the other hand, in a separate analysis of a US Air Force electronic countermeasure pod, we found severe susceptibility to false alarms due to the streamlining of available tests.[10]

Specifying additional tests

Now we will begin to use the testability analysis results to improve the testability of the system. We will focus on whether additional tests will reduce ambiguity, where these tests should be located, and what types of tests they should be. We will assume that we are conducting testing at the replaceable unit group level. Therefore, the results of the operational-isolation analysis and the ambiguity group analysis will be particularly important.

Tests on replaceable unit group outputs. As shown in Figure 1, we have defined replaceable unit groups for the case study, and we can see that some of the defined tests are placed within the group boundaries and some tests are not. Our first step will be to ensure that we test each output in the system. Therefore, we will add tests to all of the replaceable unit group outputs and label the tests ru_{n-m}, where n is the index for the replaceable unit and m is the index for the output. Figure 4 shows the dependency diagram with the additional tests, and Table 5 lists the dependencies and feeds for each new test. We leave the development of the dependency matrix as an exercise for the reader.

When we analyze testability on this new model, we see that the isolation level (IL) has increased from 0.62 to 0.77. Clearly, we still have ambiguity between



Legend:				
=Conclusion	=Test	=Testable input	=Replaceable unit group	=System input or output
21 conclusions + 4 inputs + No fault = 26	32	2	8 replaceable unit groups + 4 inputs + No fault = 13	8

Figure 4. Case study dependency diagram with additional tests.

components in the model. What we want to know, however, is whether any ambiguity exists between replaceable unit groups. The unweighted operational isolation indicates that we will fault-isolate to one replaceable unit group 65% of the time. In the weighted case, unique fault isolation will occur 99% of the time. Thus, we have substantially improved unique isolation capability, but ambiguity still exists.

If we were to compute the ambiguity table, we would find that ambiguity exists between ru_7 and inu_2, between ru_8 and inu_1, and between ru_4 and ru_5. A closer examination reveals that the first two of these ambiguities exist because the inputs are untestable. The remaining ambiguity exists because of the feedback loop, the members of which are listed in Tables 3 (conclusions) and 4.

The addition of tests to replaceable unit group outputs resulted in other interesting changes in the model. Both false failure (FF) and nondetection (ND) dropped to 0—the ideal value for both measures. False-alarm tolerance (FAT) is still very good (0.46), but the excess-test measure (XM) has increased from 0.35 to 0.62. We expected that because we added several tests without removing any existing tests.

Asymmetric and conditional tests. Despite the improvements we have made to the system, two problems still exist. A feedback loop is causing ambiguity between two replaceable unit groups, and we cannot distinguish the untestable inputs from two replaceable unit groups. Two test paradigms may be appropriate to solve these problems: the asymmetric test and the conditional test.[2,11] Because observations at inputs are frequently asymmetric, let us first examine the asymmetric test.

There are three types of asymmetric tests: fully asymmetric, positive inference, and negative inference.[11] When a positive inference test passes, we can infer that all dependencies of that test will also pass. When the test fails, we can infer no additional information. When a negative inference test fails, we can infer that all tests on which the negative inference test does not depend will either pass or be unneeded, that all feeds will fail, and that all components on which the test does not depend will be good. When the test passes, we can infer no additional information.

When we add asymmetric tests to the model, ambiguity should decrease, but we may find that the ambiguities still exist, depending on how we use the tests. We call this situation a "sometimes ambiguity." For example, let us assume that for the simple two-component and two-test system shown in Figure 5, t_1 is a positive inference test and t_2 is a symmetric test. Either we can fault-isolate c_2 uniquely (that is, t_1 passes and t_2 fails), or an ambiguity exists between c_1 and c_2 (both t_1 and t_2 fail). That is, a failure of t_1 is not enough information for us to infer anything about c_1 or c_2.

Adding asymmetric tests may make the ambiguity analysis difficult to perform because we must consider all combinations of asymmetries. For the same reason, operational-isolation computation becomes more complex. To simplify the computation, we can consider only the upper and lower bounds on operational isolation. Specifically, we can compute the upper bound simply as operational isolation, assuming the tests are all symmetric. We compute the lower bound assuming that all the asymmetric tests are eliminated from the model.

Suppose we add two tests to the model: $tasym_1$ and $tasym_2$. These tests are both negative inference tests, and they depend on inu_1 and inu_2, respectively. Because the two tests are asymmetric, we expect the lower bound on operational isolation to be unchanged. The upper bound, however, does change. First, the IL becomes 0.84 (the remaining ambiguity is due to the feedback loop). The unweighted operational isolation indicates that we will fault-isolate to one replaceable unit 85% of the time. In the weighted case, unique fault isolation

Table 5. *Replaceable unit group output test relationships.*

Test	Dependencies	Feeds
ru_{1-1}	int_1, c_3	t_2
ru_{1-2}	int_1, c_1	t_1
ru_{2-1}	t_1, c_{21}	t_4
ru_{3-1}	t_3, c_6	ru_{6-1}, t_1, t_5
ru_{3-2}	t_3, c_5	t_5
ru_{4-1}	t_6, c_8	t_7
ru_{5-1}	t_9, c_{11}	t_{10}
ru_{5-2}	t_9	t_{10}, t_{18}
ru_{6-1}	ru_{3-1}, c_{13}	None
ru_{6-2}	t_{11}, c_{15}	t_{12}
ru_{7-1}	t_4, t_{12}, c_{16}, c_{17}, inu_2	t_{13}
ru_{8-1}	t_{15}	None
ru_{8-2}	t_{14}, t_{15}, t_{17}, c_{20}, inu_1	t_{16}
ru_{8-3}	t_{17}	None

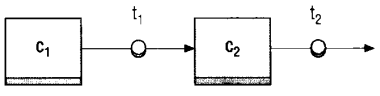


Figure 5. *Simple serial system with an asymmetric test (t_1).*

Table 6. *Test mode dependencies of conditional tests in feedback.*

Conditional test	Dependencies
t_{18}-test	t_9-test
t_9-test	c_{10}, t_8-test
t_8-test	c_9, t_7-test
t_7-test	c_8, t_6-test
t_6-test	c_7, t_5
ru_{4-1}-test	c_8, t_6-test
ru_{5-2}-test	t_9-test

will occur 99.8% of the time. Thus, we have again substantially improved unique isolation capability.

Now the only problem we have left with the system is the feedback loop. As a rule of thumb, we recommend taking care in "breaking" feedback loops. Feedback exists in a system to benefit performance. Therefore, we must develop breaks in feedback loops that do not adversely affect performance. One way to do this is to insert switches to break the loop while the system is in a test mode. But sometimes we can do nothing to break the loop. For example, a system may become unstable when the feedback loop is broken. We should then consider such solutions as repackaging. For now, we will assume that we are able to insert a switch that is available during test mode. We will associate a conditional test with this switch.

Table 7. *Redundant-test groups.*

Group	Tests
1	t_2, ru_{1-1}
2	ru_{5-2}-default, ru_{5-2}-test, ru_{4-1}-default, t_9-default, t_9-test, t_{18}-default, t_{18}-test, t_6-default, t_7-default, t_8-default
3	t_7-test, ru_{4-1}-test
4	t_{12}, ru_{6-2}
5	t_{15}, t_{16}, ru_{8-1}, ru_{8-2}
6	t_{14}, t_{17}, ru_{8-3}

A conditional test is a test whose dependencies are a function of some external condition such as user inputs, scale settings, or switches. This definition is very broad and can lead to combinatorial explosion when we analyze system testability. (Conditional dependency in sequential fault isolation is not a problem, because test selection is always conditioned on context, anyway.)

One of the best uses for the conditional test is to break feedback loops. For example, suppose we define test t_{18} in the case study to be a conditional test. Currently, t_{18} depends on t_9, which depends on t_8, which depends on t_7, which depends on t_6, which depends on t_5 and t_{18}. We have cycled back to t_{18} in the dependency chain. Let this cycle correspond to the default condition, which is the normal operational mode for the system. Thus, we can refer to t_{18} as t_{18}-default. We define conditions for t_6, t_7, t_8, and t_9: t_6-default, t_7-default, t_8-default, and t_9-default, respectively, all with the original dependencies.

Now we will define a new condition, t_{18}-test, which also depends on the tests in the feedback loop. However, to prevent completing the cycle, we define additional conditions for the original tests: t_6-test, t_7-test, t_8-test, and t_9-test, respectively. Table 6 lists the test mode dependencies for these conditional tests. Note that we have also made two of the replaceable unit group output tests conditional tests to ensure that they are available in test mode as well as default mode. Clearly, these tests will break the feedback loop because they depend on each member of the feedback loop without depending on themselves.

These new tests provide a means for uniquely isolating any replaceable unit in the system. All variations of operational isolation are 1. Unfortunately, we have added 14 replaceable unit group tests, two asymmetric tests, and seven conditional tests, more than doubling the original number of information sources. Indeed, the TL is now 1.65, indicating a high level of overtesting.

Eliminating excess tests

We have improved the testability of the missile launcher by adding 23 more tests. Obviously, with the addition of any test to the model, we must verify that the test indeed measures what we believe it measures. In any event, we will assume at this point that all the tests can be constructed and that they are correctly represented in the model. Now the question is whether we need all the tests we specified.

With a test leverage of 1.65, it seems that we do not need to develop all these tests. We will begin by eliminating the redundant tests. Table 7 lists the system's redundant-test groups. It is interesting that the tests in the now broken feedback loop continue to provide a tremendous amount of redundant information. That, of course, is because a mode in which the feedback loop is intact still exists.

In deleting the redundant tests, we must ask what criteria we plan to use for fault isolation and how the tests affect these criteria. For example, suppose t_2 takes considerably less time to perform than ru_{1-1}, but ru_{1-1} requires a lower skill level. The two tests provide identical information, so we will decide which to discard on the basis of the resources we have. If our technicians are relatively low in skill, we may want to optimize on the basis of skill level. We could then eliminate t_2. On the other hand, if time is the primary consideration, we should keep t_2 and eliminate ru_{1-1}. But the maintenance shop may have varying conditions, making it reasonable to keep both tests.

We use the excess-test analysis to determine which excess tests we can eliminate. This analysis requires us to specify optimization criteria, so the decision to delete excess tests is also sensitive to model parameters. We will assume we

are going to optimize for test times and failure frequencies. We will attempt to preserve group tests over individual tests and preserve old tests over newly created or to-be-created tests. Table 8 lists the tests recommended for removal.

When we eliminate a testable input, it becomes an untestable input. We can eliminate all the tests listed in Table 8. Although eliminating these tests has a serious negative effect on component-level testability (for example, IL drops to 62%), IL remains 100% for isolation to a single replaceable unit. In addition, both TR and XM drop to 0. Figure 6 shows the final dependency diagram for the case study, and Figure 7 (on p. 74) shows the resulting matrices, from which we derive the testability statistics.

Other issues

Although we have improved single-failure testability and reduced the number of tests, we may wish to retain some of the excess tests for other reasons. We removed redundant and excess tests without considering the effects of these actions on some subtler testability issues.

False-alarm tolerance. Let us examine the impact of our actions on one of these issues—our ability to detect false alarms. One way to detect a false alarm is to use tests with inferred outcomes to cross-check the evaluated tests. In our March article, we defined false-alarm tolerance as a measure of our ability to perform test-to-test cross-checking. Obviously, as we remove excess tests, we should expect FAT to decrease. That has indeed happened. As we can see from Figure 8a, FAT decreased each time we modified the model. In fact, FAT for the last model is half that for the original model. In general, the linear relationship may not hold, but the message is clear: Tests should be removed sparingly if the system has potential false-alarm problems. In this case, FAT = 0.24, which is still relatively good.

Multiple failures. Another issue we should consider when removing excess tests is our ability to fault-isolate in the presence of multiple failures. As tests are removed, the number of hidden failures may increase because we could have used the eliminated tests to make the failure signature (discussed in the March article) distinct from other failures. Because false failures are related to hidden failures, we can expect false failures also to increase.

Table 8. *Excess tests recommended for elimination.*

t_1	t_6-default	ru_{8-3}
t_3	t_6-test	ru_{3-1}
t_5	t_7-default	ru_{3-2}
t_{10}	ru_{4-1}-test	ru_{4-1}-default
t_{11}	ru_{4-1}-default	ru_{8-1}
t_{13}	t_8-default	ru_{8-2}
t_{14}	t_8-test	ru_{8-3}
t_{15}	t_9-test	ru_{6-2}
t_{17}	t_9-default	ru_{1-1}
int_1	t_{18}-test	
ru_{2-1}	t_{18}-default	

Removing redundant tests has no effect on either the hidden-failure measure (HF) or the false-failure measure (FF) because redundant tests provide us no additional capability to identify either single or multiple failures in the system. Thus, there is no change in HF or FF between the model with all the additional tests and the model with the redundant tests removed. It follows that we could have mathematically collapsed the redundant tests in the mod-

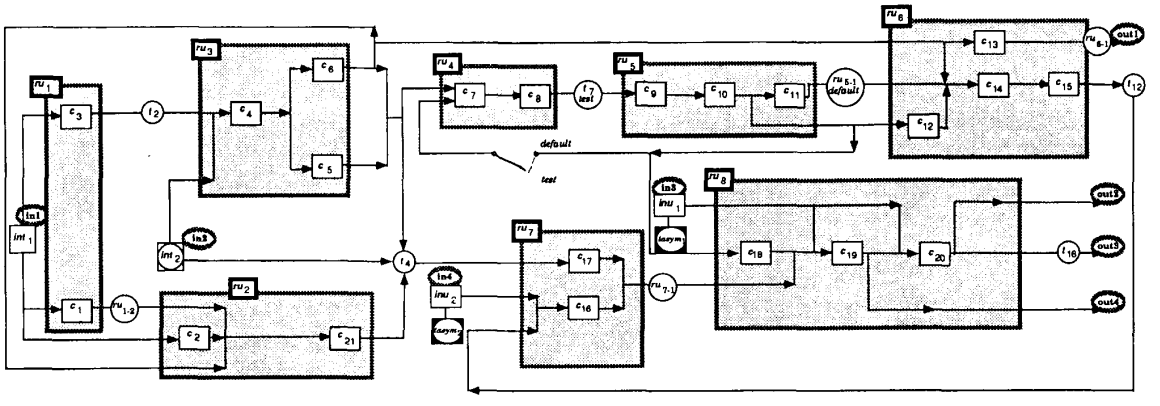
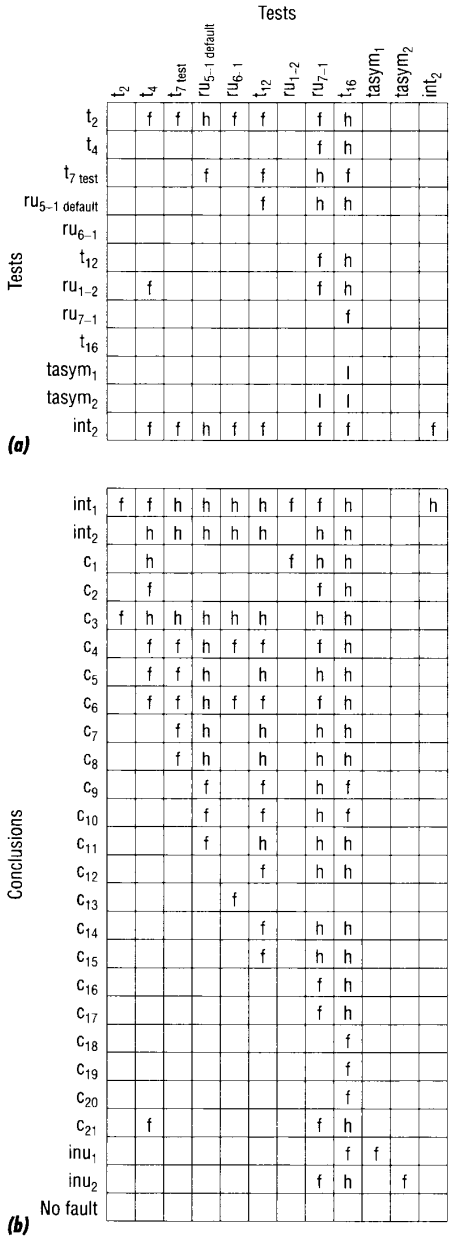


Figure 6. *Case study dependency diagram after testability analysis.*



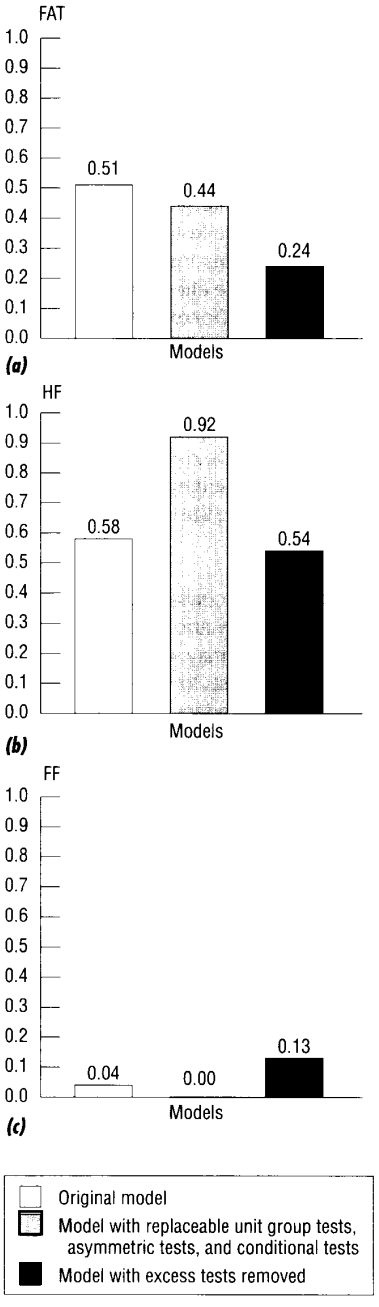


Figure 7. *Closed dependency matrices after testability analysis: test-to-test* ***(a)****; test-to-conclusion* ***(b)****.*

Figure 8. *Variations in* ***(a)*** *false-alarm tolerance (FAT),* ***(b)*** *hidden-failure (HF), and* ***(c)*** *false failure (FF) measures.*

el. In fact, that is exactly what happens during fault isolation (to be discussed in the next article in this series).

Removing excess tests, however, does affect both HF and FF. In fact, as Figure 8b shows, HF has decreased with the remov-

al of excess tests. However, HF for the model with excess tests removed is still approximately equal to HF for the original model. More significantly, FF (Figure 8c) increased from 0 to 0.13, greater than FF for the original model.

If we examine the subsignatures of the fault isolation conclusions, we find two faults with potential false indications. Examining the model, we see that multiple failures of c_1 and c_3 will look like a failure of int_1, and multiple failures of c_5, c_{13}, and c_{21} will look like a failure of c_4 (or c_6, because c_4 and c_6 are ambiguous). To determine if these potential false indications are significant, we must determine the probabilities that the multiple failures will occur. If the probabilities are high, we must take some action, either adding tests or restoring some of the eliminated excess tests. If the probabilities are low (or the multiple failures occur in the same replaceable unit group as the single failure), no action may be necessary.

To examine the probabilities of these multiple failures, we will consider the failure rates and ignore the high probability of *No Fault* (because we assume that a fault has occurred). We also assume that the failures are independent, so the probabilities of the multiple failures are the products of the individual probabilities. (This analysis fails completely if the failures are not independent events. Any interdependency would indicate that we cannot tolerate the false-failure situation. Interdependency could come from a root cause situation, in which one failure actually causes the other, or from an increased-stress situation, in which the failure of x changes the failure rate of y.) Table 9 lists the probabilities for the components of interest (derived from the weights given in our March article).

Because c_2 and c_{21} are ambiguous, we must consider the possibility that either fails to contribute to the false indication of c_4. The probability of a false indication of int_1 is

$$P(f = c_1)\,P(f = c_3) = (0.0055)(0.1105) = (0.0006),$$

which is an order of magnitude less than the lowest single failure. Therefore, we will not worry about the occurrence of the multiple failure. Making int_1 testable again will eliminate the false-indication problem.

We determine the probability that c_4 or c_6 will be falsely indicated by a multiple failure in a similar manner:

$$[P(f = c_2) + P(f = c_{21})]\,P(f = c_{13})\,P(f = c_5) = (0.0055 + 0.1105)\,(0.0055)\,(0.1105) = (0.00007).$$

This probability is *two* orders of magnitude less than the lowest single failure. Therefore, we will not worry about the occurrence of these multiple failures, either. Our third article explains in detail how to include multiple failures in the model itself to avoid the false-failure situations.

Alternative approaches

Thus far we have focused on techniques for improving system testability with available test resources. Now we will discuss two alternative approaches to improving testability: redesigning the system and repackaging functional elements of the system.

Redesign. Previously, we assumed that the physical design of the system being analyzed was fixed. However, if the system is in the early phases of its life cycle, it may be possible (and even cost-effective) to improve the design from a testability perspective. For example, suppose the target tracker of the current system uses a radar-based sensor. If we are having trouble detecting certain failure modes associated with the radar-based tracker, we might consider using a laser tracker instead.

Of course, if we redesign the system, we must consider several issues. For example, the cost of redesign may be prohibitive. Laser sensors may be an order of magnitude more expensive than radar-based sensors, so we may be willing to accept the degraded testability. Further, a design is often optimized for performance. If a redesign adversely affects performance, it is probably unacceptable. Also, the new tests associated with a new design may cause unanticipated testability problems such as higher probability of false indication or degraded ability to detect false alarms. In fact, the new design may be prone to false alarm, and the tests may be oversensitive, thus compounding the problem.

Redesigning a system for testability is extremely complex; it essentially takes the design process back to the early stages. We usually can minimize redesign problems by making testability a principal input at each step of system design. But when a complete redesign is impractical because testability has not played a key part in the design process, we can limit the scope of redesign through functional repackaging.

Repackaging. Given the design of a system, the way the functions of the design are packaged may directly affect the system's testability. This became evident earlier in this article, before we made any changes to the testability of the case study system. Recall that we assigned several components to different replaceable unit groups. These groups corresponded to the

Table 9. *Probabilities of component failure.*

Component	Probability
c_1	0.0055
c_2	0.0055
c_3	0.1105
c_5	0.1105
c_{13}	0.0055
c_{21}	0.1105

system's functional packaging. When we examined operational isolation, we found that the system exhibited tremendous ambiguity between the replaceable units. Two contributors to ambiguity were the nondetection of c_{13} and the existence of two untestable inputs. Obviously, no level of repackaging will solve these problems, but repackaging could have corrected other ambiguities in the system.

The ambiguity groups for the original model are listed in Table 3. We can see that the ambiguity of c_1 and c_2 results in ambiguity between ru_1 and ru_2. This is not good. Because neither c_3 nor c_{21} is ambiguous with any other components, we may want to combine ru_1 and ru_2, thus completely enclosing the ambiguity in one group (say, $ru_{1\&2}$). We can do exactly the same thing for the members of Group 2 because they make up the feedback loop. However, we find that since c_{11} is in the now-combined group (say, $ru_{4\&5}$) and c_{12} is in ru_6, we may want to repackage by putting c_{11} in ru_6 instead, thus preventing ambiguity between $ru_{4\&5}$ and ru_6. Because c_{11} and c_{12} are now in the same replaceable unit group, we no longer have an ambiguity problem in Group 3. However, c_{13} in Group 4 is still undetectable. The only way to solve this problem is by adding at least one test.

We can treat Groups 5 and 6 together. First, we can consider the inputs as part of the replaceable unit groups with which they are ambiguous rather than as individual components. This would solve the ambiguity problem for Group 6. However, it may be more appropriate to treat Groups 5 and 6 separately, in which case we need to define additional tests. In addition, Group 5 creates ambiguity between ru_7 and ru_8. A solution would be to repackage c_{18} in ru_7.

To summarize, we could repackage the case study as follows:

- Combine ru_1 and ru_2 into $ru_{1\&2}$.
- Combine ru_4 and ru_5 into $ru_{4\&5}$.
- Repackage c_{11} as a member of ru_6.
- Repackage c_{18} as a member of ru_7.
- Add tests to directly observe inu_1, inu_2, and c_{13}.

After making these changes to the original model, we find that TR decreases from 0.3 to 0.26, XM decreases from 0.35 to 0.3, and all the operational-isolation measures become 1. As a result, we need not worry about adding a large number of tests to the system, and we can concentrate on other issues, such as false alarms and multiple failures.

Extending testability analysis

We have been examining issues directly related to system testability to determine our ability to perform fault isolation. Now let us examine several additional applications of these testability analysis techniques.

Operational-readiness inspection. Much of the testing performed by the military services takes place during operational-readiness inspections. The purpose of an ORI is to ascertain if a weapon system is healthy and functioning and, when a material problem exists (such as a poor radar video output), to localize its source sufficiently to take action. The overall goal is to minimize test resources and provide maximum coverage.

The tests performed by ORI teams correspond directly to tests in the information flow model. Each test, as an information source, has a set of dependencies. If the test passes, the set of dependencies or elements is verified. If the test fails, some member of the dependency list is suspect. We can derive the corresponding model by using the details of the tests available and the weapon system schematics in the same manner as a normal fault isolation analysis. Using the outcomes of asymmetric, conditional, and linked tests is also appropriate.

A key difference between modeling for ORI testing and modeling for normal testability is in the setup of test elements. Constraints in the operational environment may cause the analyst to place a number of tests into a forced group. This means that all tests will be evaluated (that is, inference is suspended.) Such constraints may also cause the analyst to sequence groups in a special order by various methods (weighting or direct sequencing). The analyst may also anticipate multiple failures by checking the staffing levels for each system. For example, if the video technician is new, the maintenance of the system may be suspect, and the analyst may want to include multiple failures in the video hardware areas for the analysis. Despite these restrictions, ORI analysis can provide significant improvements in testability.[12]

In addition to diagnostic or readiness testing, the information flow model can serve as a logic model for knowledge base verification.

Knowledge base verification. In addition to its use for diagnostic or readiness testing, the information flow model can be used as a logic model for knowledge base verification.[13,14] Specifically, we can use a model of the knowledge base to obtain indications of knowledge base consistency, completeness, and correctness. If we assume that our knowledge base is a rule base, we can map rule antecedents to tests and we can map rule consequents to model conclusions. We can then easily determine dependency relationships by examining the chains of inference through the knowledge base.

Consistency analysis consists of iden-

tifying problems of overspecification and inappropriate specification of the rules. Results include identification of redundant rules, subsumed rules, redundant *if* conditions, and logical circularity. The information flow model actually absorbs redundant and subsumed rules, and we can use the model to derive a reduced set of rules. We identify redundant *if* conditions by locating redundant and excess predicates (tests). We identify logical circularity directly from the feedback analysis.

Completeness analysis is similar in principle to consistency analysis in that we are attempting to determine whether the specified set of rules is sufficient for us to draw the required set of conclusions. The parts of the testability analysis related to completeness include identifying ambiguity, identifying conclusions that will never be drawn (called dead-end goals), identifying *if* conditions that do not lead to a conclusion (called dead-end *if* conditions), and anticipating the effects of illegal attribute values. Completeness analysis uses the failure-modes-and-effects analysis (FMEA) that arises from the information flow model.

Correctness analysis examines specific inference traces and drawn conclusions. The most important element of a correctness analysis is a domain expert, which examines inference traces and results to determine if the drawn conclusions are correct. We can perform parts of the correctness analysis from the FMEA, but we need an inference engine to perform a thorough analysis. The fault isolation analysis we will describe in future articles in the series is most applicable to the correctness issue.

Software testability. The focus of the integrated diagnostics concept is on system-level testability. For most systems, testability must address not only hardware but software. The majority of the analyses available from the information flow model address hardware testability and are difficult to apply to software. Nevertheless, we have developed model extensions and modeling techniques for high-level software analysis.[15,16] At the functional level, we can model software functions and tests in much the same way as for hardware testing. Unfortunately, software testing is philosophically different from hardware testing, so information flow modeling is not amenable to low-level software testability.

Software testability requires testing a system design. Software does not fail; rather, design flaws become evident as different paths through the software are traversed. Zero-defect software cannot be achieved because exercising all possible paths and all possible states in an arbitrarily complex software system is computationally impossible. Further, if a bug is identified and corrected, testing must essentially start from scratch (the principle behind regression testing) because the design of the software has now changed. If the information flow model has too high a resolution, the fix will require a modification of the model. Modifying the model can be cumbersome and is generally not feasible.

Performance testing. Another concern in integrated diagnostics is to determine whether a system is performing according to the design specifications. The resulting test scenarios are related not to fault diagnosis but to performance evaluation. Operational readiness inspection addresses some of the problems related to performance, but another aspect of operations (performance) testing is the operational-readiness evaluation (ORE). ORE differs significantly from ORI in that ORE evaluates personnel and procedures, together with equipment. In addition, such concerns as timing, efficiency, and applicability are pertinent to system performance. We have not yet attempted to represent performance issues in the information flow model. We believe that modeling system performance (at least to a point) is possible, but this problem is similar to and even less well constrained than software testing.

IN THIS ARTICLE AND THE PREVIOUS one, we have focused on applying the information flow model to assess system testability and optimize available test resources. We have examined the problems of excess information provided by a test set, ambiguity arising from deficiencies in the test set, multiple failure, and the test set's effect on false-alarm tolerance. We have applied the analysis techniques to a specific example—the antitank missile launcher—and have demonstrated how we might improve the testability of that system.

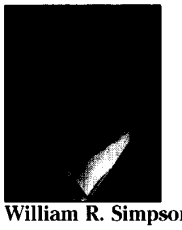
Improving system testability is one step in the integrated diagnostic process. The next step we will examine is using the available resources to isolate faults. In particular, we will be interested in deriving the best set of strategies for the specific conditions of the system. We need to consider inference methods, hierarchical inference, multiple-attribute tests, optimization criteria, and consistency checking. The next article will provide the algorithmic details of fault isolation, and the following article will provide a detailed example of isolating faults in the case study system. D&T

References

1. W.R. Simpson and J.W. Sheppard, "System Testability Assessment for Integrated Diagnostics," *IEEE Design & Test of Computers*, Vol. 9, No. 1, Mar. 1992, pp. 40-54.
2. J.W. Sheppard and W.R. Simpson, "A Mathematical Model for Integrated Diagnostics," *IEEE Design & Test of Computers*, Vol. 8, No. 4, Dec. 1991, pp. 25-38.
3. *Testability Program for Electronic Systems and Equipment, MIL-STD-2165*, Naval Electronics Systems Command (ELEX-8111), Washington, DC, 1985.
4. D. Droste and W. Parsons, "Testability: The State of the Art—Automatic Testing in the Next Decade and the 21st Century," *Autotestcon 89 Conf. Record*, IEEE, New York, 1989, pp. 168-174.

5. W.L. Keiner, "A Navy Approach to Integrated Diagnostics," *Autotestcon 90 Conf. Record*, IEEE, New York, 1990, pp. 443-450.
6. J. Franco and J. Scott, "WSTA—The IDSS Weapon System Testability Analyzer," *Autotestcon 86 Symp. Proc.*, IEEE, New York, 1986, pp. 435-440.
7. K. Pattipati and M. Dontamsetty, "Studies in Testability Optimization," Mid-West Testability Group, Mulville, New York, 1988.
8. C.R. Unkle and H.P. Himpler, "STAMP Demonstration for Air Pressurization System," letter report, Arinc Research Corp., Annapolis, Md., 1985.
9. D.C. Curtis et al., "AN/MSQ-103C Teampack Testability and Fault Isolation Analysis," Publication 2968-01-4009, Arinc Research Corp., 1986.
10. E. Esker and H. Horvath, "System Testability and Maintenance Program for the AN/ALQ-285(V) (Seek Ice Modification Program)," Arinc Research Corp., 1985.
11. W.R. Simpson and J.W. Sheppard, "System Complexity and Integrated Diagnostics," *IEEE Design & Test of Computers*, Vol. 8, No. 3, Sept. 1991, pp. 16-30.
12. B. Pickerall, "Testability Analysis and Revised Procedures for Overall Combat System Operability Test (OCSOT) USS Dale (CG-19)," Tech. Report 3599-01-01-4528, Arinc Research Corp., 1987.
13. J. Sheppard, "An Approach to Verifying Expert System Rule Bases," Artificial Intelligence Tech. Note 1700, Arinc Research Corp., 1989.
14. J. Sheppard, "An Approach to Rule-Base Verification," Artificial Intelligence Tech. Note 1322, Arinc Research Corp., 1989.
15. J. Sheppard and W. Simpson, "Functional Path Analysis: An Approach to Software Verification," *Proc. ACM 16th Annual Computer Science Conf.*, ACM, New York, 1988, pp. 266-272.
16. R. Bond and J. Sheppard, "Structural Analysis Methods to Aid in Software Testing, Debugging, and Maintenance," Software Engineering Tech. Note 2402, Arinc Research Corp., 1989.

John W. Sheppard is a senior 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 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. Sheppard 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 to the authors at Arinc Research Corp., Advanced Research and Development Group, 2551 Riva Rd., Anapolis, MD 21401; e-mail: sheppard @cs.jhu.edu or wsimpson@ mcimail.com.