Requirements-Based Testing in Aircraft Control Design

Jason Ghidella* and Pieter J. Mosterman†
The MathWorks, Inc., Natick, MA, 01760, USA

To be competitive, Model-Based Design can be applied to help bring down the cost of system design and faster time to market. Model-based approaches are especially effective in the design stages and are increasingly tied in with the requirements capture and code generation and testing phases. The infrastructure for this allows linking requirements to parts of the system model as well as automatically generating references to the original requirements in the code. In addition, test vectors that are derived from the requirements are part of the system model and can be linked to the requirements they represent. This paper describes how this infrastructure can be combined with coverage analyses for model verification and validation to aid in making requirements consistent and unambiguous, while ensuring the set of test vectors is complete and the design minimal (i.e., no superfluous elements exist).

I. Introduction

MODERN aerospace control systems have reached a level of complexity that requires systematic methods for their design. Typically, system development begins with the gathering of high-level requirements, which are stored in text format. These requirements form the basis for the system specification, which is gradually refined into a detailed design that can be implemented. During implementation, engineers use structured test scenarios to validate system behavior against the original requirements.

With Model-Based Design,[1] designers use graphical models, often a block diagram, to capture requirements. They produce an executable specification that can be gradually extended into an increasingly detailed design from which the implementation code can be automatically generated. The validation process can take place much earlier in the overall system design effort, reducing costly iterations across many design steps.

To efficiently exploit the advantages of Model-Based Design, the control design tool infrastructure must be able to relate the requirements to the system behavior. In this context, the system behavior is determined by predefined excitation, called test vectors, and certain characteristic system dynamics. All of the system scenarios require test vectors in order to establish the presence of the desired behavior as defined by the requirements. It is also critical that the system does not exhibit additional or undesired behavior, which is tested by exhaustively exciting the system design. To achieve these two verification goals, engineers employ coverage analyses such as decision coverage, which establishes whether logical decisions have been evaluated true and false during the course of testing.

This paper discusses the use of Commercial Off-the-Shelf (COTS) Software with Model-Based Design to develop a fault detection, isolation, and recovery (FDIR) application for a redundant actuator control system. This example will demonstrate how requirements can be associated with semantic elements of the design as well as the input test vector set. The characteristic system dynamics (both desired and undesired) are incorporated as model assertions. In turn, these model assertions are separately related to individual test cases within the test vector set, and also related to the requirements. To ensure the absence of undesired system behavior, full coverage analysis is performed on the design model. Similar coverage analyses are typically conducted on the implementation code.

This scenario demonstrates the benefits of coverage analysis, particularly the ability to conduct test and verification early in the design. For example, ambiguities in the written requirements are detected

*Technical Marketing Manager, 3 Apple Hill Dr., Natick, MA 01760, AIAA member.
†Senior Research Scientist, 3 Apple Hill Dr., Natick, MA 01760, AIAA member.

and corrected as the design is developed. Left undetected until a design is complete, poorly interpreted specifications can result in design errors that are costly and difficult to eradicate. In addition, coverage analyses identify three types of undesired design characteristics: omissions in the test vector set (indicated by parts of the design that do not execute), design elements that are not tied to the requirements and are deemed superfluous, and inconsistent, ambiguous, or lacking requirements.

Section II gives an overview of the system under design, an actuator redundancy control system. In Section III, it is discussed how the high-level requirements need to be translated into more detailed ones. Section IV shows how requirements can be related to assertions and tests. In Section V, some model coverage results for the actuator redundancy control system are given. Section VI shows how the coverage results can be used to modify the test vector set and how they may lead to design changes. Section VII shows how certain requirements may have to be modified or added. In Section VIII the coverage results after the test, design, and requirements modifications are presented. Section IX presents the conclusions of this work.

II. The System

THE design of the fault detection, isolation and recovery (FDIR) system for a redundant elevator control system was described in detail in.[2–5] In this work, the steps to verify and validate that design, specifically, the mode logic component of that system are considered.

Figure 1 shows a simplified configuration of the redundancy typically found in an aircraft elevator system for civil aviation. It consists of two elevators, one on the left and one on the right. Each elevator can be positioned by two actuators, only one of which should be active at any given time. The four actuators are connected to three separate hydraulic circuits, as shown in Fig. 1. The two primary flight control units (PFCU) are used to control either the inner or outer actuators. To this end, there are two control laws available. In nominal operation, a sophisticated *input-output* (IO) control law is applied to the left (LIO) and the right (RIO) outer actuators. In case of a failure, a *direct link* (DL) control law with reduced functionality is available for the left (LDL) and right (RDL) inner actuators.

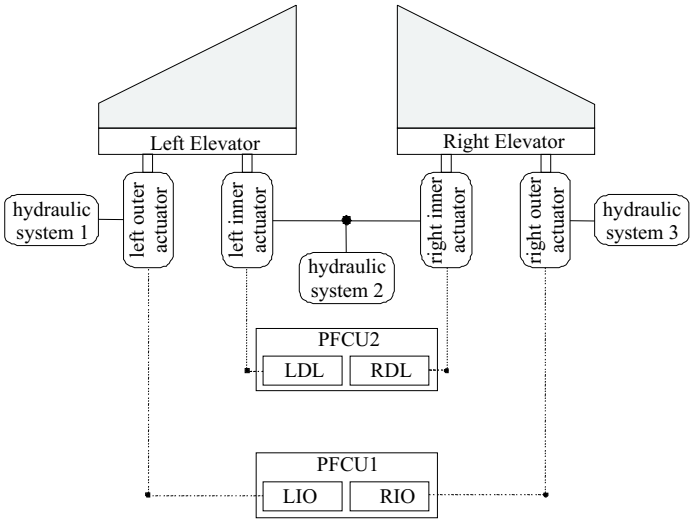
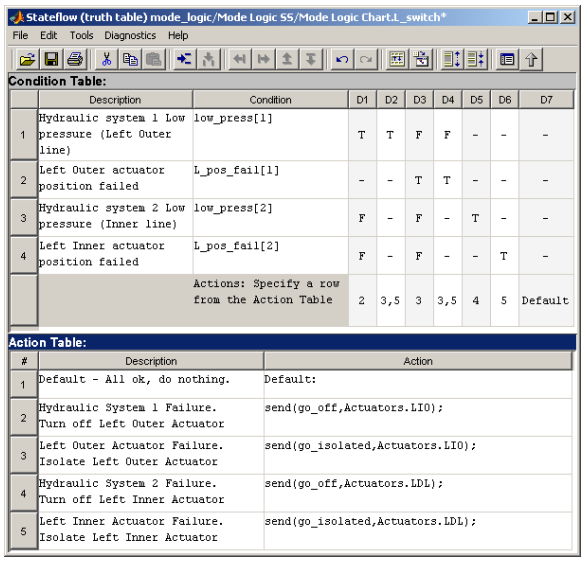


Figure 1. The elevator redundancy configuration.

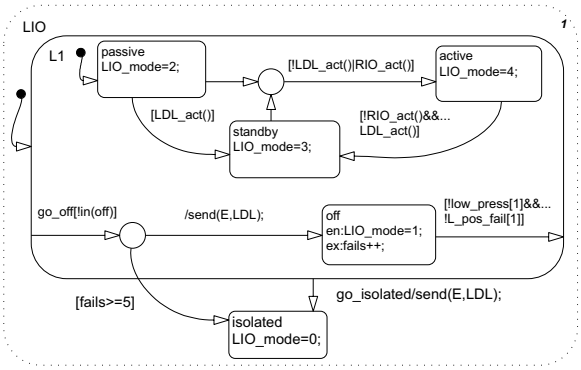
The example in Fig. 1 relies on *physical redundancy*: When one component fails, another can be activated, and it is the mode logic that determines when and how this should occur.

The mode logic consists of two components. The first tries to isolate the fault by looking at the symptoms that have been detected. This is done in a truth table shown in Fig. 2(a). The truth table consists of two parts, a *condition table* at the top and an *action table* at the bottom. The conditions in the "Condition" field of each of the rows are evaluated for each of the decisions in the columns marked "D1" through "D7" from left to right. The action taken is the one that corresponds to the first decision column with truth values selected that match the logic values of the conditions. For example, assume the situation that in Fig. 2(a) *low_press[1]* is false, *L_pos_fail[1]* is true, *low_press[2]* is true, and *L_pos_fail[2]* is false, which corresponds to the logical combination [!c1&c2&c3&!c4]. First, D1 is evaluated against these logic values, and because *low_press[1]* is false, the corresponding action, 2 (indicated at the bottom of the column), is not taken. Next, D2 is evaluated and its actions, 3 and 5, are not executed for the same reason. Then, D3 is evaluated, but because *low_press[2]* is true, its action, 3, is not selected. Finally, D4 has all its conditions satisfied (the '-' entry indicates either logic value is acceptable for that condition) and actions 3 and 5 are executed in that order. In the "Action Table", these actions are shown to be *send(go_isolated,Actuators.LIO)* and *send(go_isolated,Actuators.LDL)*, which sends the event *go_isolated* to the left outer and inner actuator switching logic modules. The truth table shown in Fig. 2(a), governs both left side actuators, a similar truth

table has been constructed to isolate the faults detected on the right side actuators.



(a) Truth table logic.



(b) Switching logic.

Figure 2. The left IO module logic.

The second component of the mode logic is the switching logic that recovers from the isolated fault. This is best handled in a state transition diagram, shown in Fig. 2(b). This diagram shows the five possible actuator states, *isolated*, *off*, *passive*, *standby*, and *active*, for the left outer actuator. Upon initialization, the diagram moves into the *passive* state and the system status is checked to determine whether the actuator should become *active* or move to *standby*. When faults are observed, the actuator may go into the *off* or *isolated* state, depending on the nature of the fault and whether it will be recoverable, or not, respectively. From the truth table example, the event *go_isolated* was sent to the left IO module (as well as the left DL module). From Fig. 2(b), it can be seen that the state transition diagram transitions from the *L1* state to the *isolated* state. As such it is possible to isolate the actuator regardless of the particlar state within *L1* it was operating.

The switching logic shown in Fig. 2(b) is for the left outer actuator, similar state transition diagrams are constructed for the left inner, and right outer and inner actuators. The switching logic for each actuator is independent but is inherently tied together, as the actuator mode of one directly affects the actuator modes of the others.

III. Towards Detailed Requirements

In general, there are a variety of requirements for the redundancy management.[5] They typically are formulated in natural language and may, indeed, contain inconsistencies. In this paper, a small set of high-level requirements, listed in Table 1, is used to guide the design of the mode logic, the actual requirements are far more extensive than those presented.[2,3,5]

Natural language requirements such as those in Table 1 can be incomplete, inconsistent, and difficult to interpret. The combination of Requirement 2 and Requirement 3 is an example of the possible ambiguity: If one actuator can be operated only in DL, should the other still be operated in IO? In other work,[4] Stateflow®[6] was used to formalize the desired behavior and uncover and resolve many of the issues through normal modeling and simulation tasks.

To design an implementation, the high-level system requirements need first be translated into more detailed, subsystem level, requirements. To illustrate, Requirements 2, 3 and 4, as specified in Table 1, suggest the following behavior in the design:

If no failures have been detected, then the left and right outer actuators should be *active*, while the left and right inner actuators should be in *standby*.

Part of the detailed requirements that can then be translated into (formal) specifications is shown in Table 2. Based on these specifications, a controller model is designed that embodies the desired behavior. The design then needs to be tested against those requirements to verify and validate its compliance. Producing the detailed requirements and formal specifications tends to be a complex process, and typically errors in the design and requirements are not found until late in development where they are difficult and expensive to address.

The approach demonstrated in this paper revolves around being able to independently establish test cases that are derived from the high and low-level requirements that can be executed on the design model, that has been created based on the same requirements, to verify that the requirements have been met. While those tests are being executed, it is important to collect coverage metrics on the design model (giving measurable evidence of the tests). Additionally, but not covered in this paper, is the ability to have full traceability of the high and low-level requirements to the design model, and through to software implementation.

Table 1. Elevator system requirements.

ID	Description
1	Each actuator will have 5 modes; *isolated*, *off*, *passive*, *standby*, *active*.
2	If possible the same control law should be *active* for both the left and right elevators.
3	If available, the IO control law should be *active* instead of the DL control law.
4	The actuator that is not active should be in *standby*.
5	If the pressure of the hydraulic circuit is low and the position measurement fails, the corresponding actuator should be switched to *off*.
6	If the pressure of the hydraulic circuit is nominal and the position measurement fails, the corresponding actuator should be switched to *isolated*.
7	Controller state changes should be made only in response to failure events.

Establishing such a formalized approach to testing the design is important to demonstrate that it meets the written requirements. It is also necessary to quantify how much of the design has been tested, to document this information, and gain measurable confidence in the design.

IV. Requirements, Tests, and Assertions

To test the requirements, a test harness is established in Simulink®[7] (see Fig. 3), where input is created in the Signal Builder block, *Test Cases*, and system output is checked via assertions specified by verification blocks in *check modes*. Using Simulink Verification and Validation,[8] each test case in the Signal Builder block can be associated with specific verification blocks, so as to check for the expected output for that specific test. Additionally, each test case can be associated with a requirement that it is testing.

To illustrate, Requirement 2.2.1 from Table 2 describes the "no failure" of the nominal operating condition of the system. To test that the design is able to perform this task, the Signal Builder is used to design the test case shown in Figure 4(a). The inputs for all the failures are set to zero, representing that no failures have been detected.

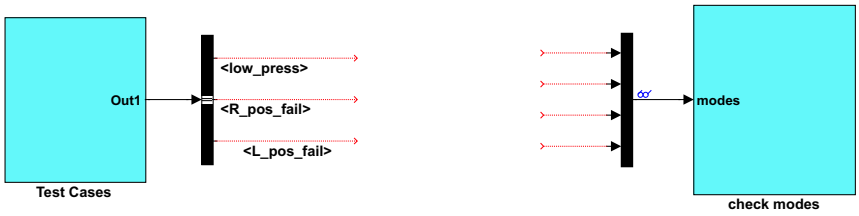


Figure 3. Test harness to test mode logic design.

The additional input signal labeled "do_verification" at the bottom of the list of signals in Fig. 4(a) is used to enable the verification blocks to start checking system output, once the system is in the correct configuration. For example, for the third test case, labeled "H1 Failure Recover" (see Requirement 2.1.2 in Table 2), the hydraulic line must first fail, and then come back online. It is only when the pressure returns to normal that the actuator modes should be checked whether they are in the expected configuration.

On the right-hand side of the dialog for the Signal Builder block in Fig. 4(a), there are two panes, one

Table 2. Partial list of the detailed requirements for the mode logic design.

2.1.1 Hydraulic pressure 1 failure
If a failure is detected in the hydraulic pressure 1 system, while there are no other failures, isolate the fault by switching the Left Outer actuator to the off mode.
2.1.2 Hydraulic pressure 1 fails and then recovers
If a failure is detected in the hydraulic pressure 1 system and the system then recovers, switch the Left Outer actuator to the standby mode.
2.1.3 Hydraulic pressure 2 failure
If a failure is detected in the hydraulic pressure 2 system, while there are no other failures, isolate the fault by switching the Left Inner actuator and the Right Inner actuator to the off mode.
2.1.4 Hydraulic pressure 2 fails and then recovers
If a failure is detected in the hydraulic pressure 2 system and the system then recovers, switch the Left Inner actuator and the Right Inner actuator to the standby mode.
2.1.5 Hydraulic pressure 3 failure
If a failure is detected in the hydraulic pressure 3 system, while there are no other failures, isolate the fault by switching the Right Outer actuator to the off mode.
2.1.6 Hydraulic pressure 3 fails and then recovers
If a failure is detected in the hydraulic pressure 3 system and the system then recovers, switch the Right Outer actuator to the standby mode.
2.2.1 Default start-up condition
If there have been no failures detected, the Outer actuators have priority over the Inner actuators. Therefore the elevator actuators should default to the following modes. The Left Outer and Right Outer actuators should transition from the Passive mode to the active mode. The Left Inner and Right Inner actuators should transition from the Passive mode to the Standby mode.

labeled "Verification block settings" and the other "Requirements". It is through these two panes that each test case can be associated with specific verification blocks and requirements that the test case has been constructed for.

The "Verification block settings" pane shown in more detail in Fig. 4(b) presents a filtered view of the Simulink model, showing the hierarchy of all the included verification blocks. For the selected test "no failures", assertions have been enabled to check the expected system output for the left inner and right inner actuators to be in *standby* mode, while the left outer and right outer actuators are to be in the *active* mode.

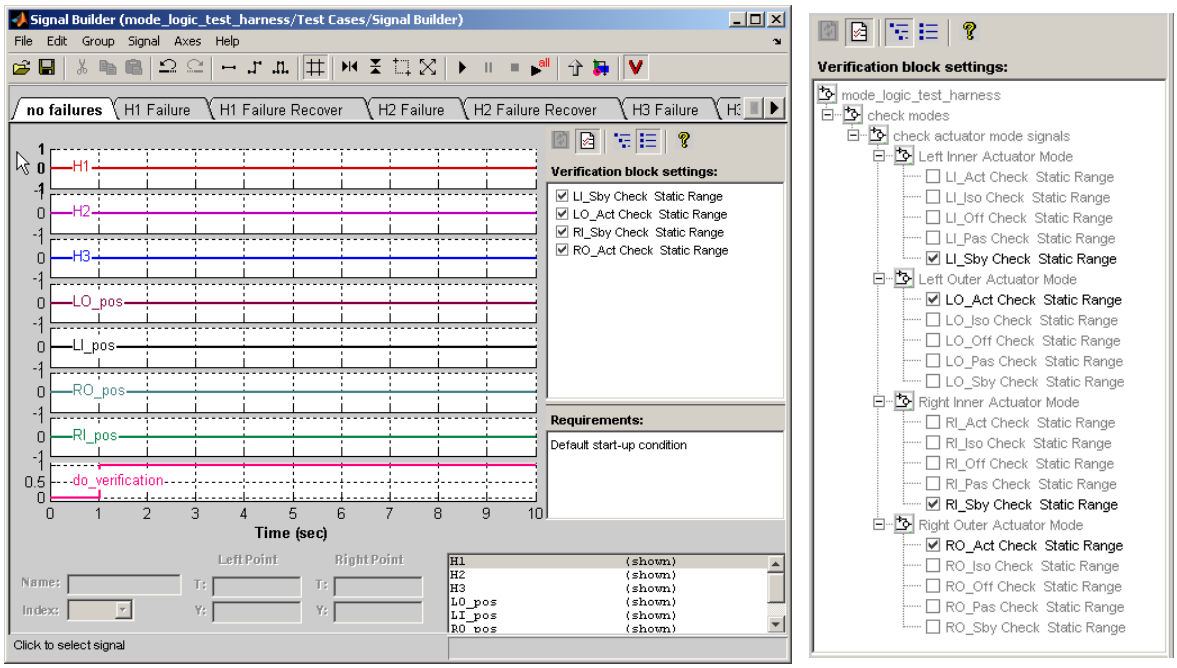
Similarly, the "Requirements" pane shows the requirements that are associated with this test case. The requirement description states that this is the "Default start-up condition". Double-clicking on this text navigates to the particular section in the requirements document that describes this requirement shown in Table 2 in detail.

Requirement associations are created and edited through a graphical user interface (GUI), as shown in Fig. 5. This GUI allows selecting the document type, the actual document, and the requirement location identifier. The location in the document that refers to the default start-up condition requirement as defined in Table 2 can be defined by selecting the desired requirement from a list of document headings and bookmarks that are automatically generated in the "Document Index" tab.

A similar approach was taken to establish test cases for each of the detailed requirements. A list of some of these requirements is shown in Table 2. In total, 23 test cases were created, and all were associated with verification blocks that checked for expected outputs of the design and requirements for which the tests were created.

The design can be tested by incorporating the mode logic subsystem into the test harness in Fig. 3 to form a testing model.

Tests can then be executed one at a time, or in batch mode. If a test runs through to completion, without any verification blocks asserting, then the design passes the test. If an assertion is detected, the simulation is stopped and the verification block that issued the assertion is highlit, allowing quick diagnosis of why the test did not pass.



(a) Test vectors for the *no failures* test case.

(b) Verification block settings.

Figure 4. Associating requirements, assertions, and tests.

V. Coverage

CREATING tests based on the requirements and executing those tests to ensure the design behaves as expected is not sufficiently rigorous to guarantee the design has been fully tested: (i) requirements could be lacking tests, (ii) the requirements themselves could be ambiguous and incomplete, and (iii) the design could contain superfluous elements.

Using Simulink Verification and Validation,[8] coverage metrics are collected as the tests execute to quantify which elements of the design have been excited and to focus attention on why the remaining elements have not. These coverage metrics are displayed directly on the model using color schemes and highlighting. Additionally, a detailed report of the coverage analysis is created. Coverage is enabled through a GUI, allowing selection of which coverage metrics will be reported for which subsystem of the model.

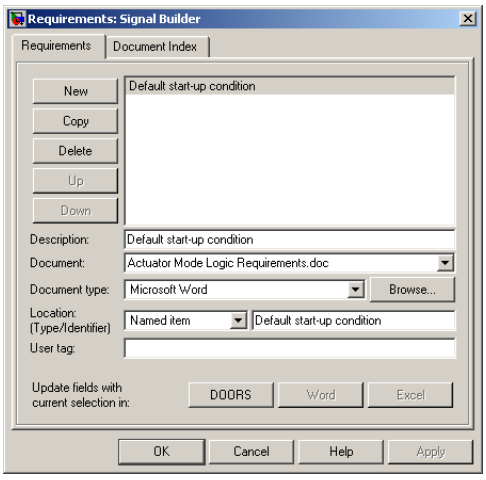


Figure 5. The requirements UI.

In this work, a number of metrics consisting of (i) *cyclomatic complexity* and (ii) coverage will be collected on the Mode Logic subsystem. The *cyclomatic complexity* is an approximation of the complexity of software (e.g., nested decision points)[9] which is adapted to models.[8] The *Decision Coverage* analysis reports on which logical decisions in the model have been evaluated to true and false. The *Condition Coverage* analysis reports on which conditions have been evaluated to true and false. The *Modified Condition, Decision Coverage* (MC/DC) is essential for DO-178B[a] certification and requires the execution of each separate input to a logical expression that can independently affect the outcome of the decision, while the other conditions are held constant. Not used in

[a] http://www.rtca.org

this paper are *Look-up Table Coverage* which reports on what entries of a look-up table and which interpolation intervals have been excited, and *Signal Range Coverage* which provides minimum and maximum values for the signals in a model.

Executing each of the 23 tests yielded the coverage report shown in Fig. 6. The coverage report reveals that even though tests were created for all the detailed requirements, the design has not been fully excited indicated by the fact that full coverage has not been achieved.

As can be seen in Fig. 6, decision coverage (column **D1**) is the easiest metric to achieve, where no subsystem of the design achieved less than 89% coverage. On the other hand, condition coverage (column **C1**) and modified condition, decision coverage (column **MCDC**) are more restrictive, with some subsystems of the design having coverage as low as 78% and 56% respectively.

The coverage results for the truth tables *L_switch* and *R_switch* gained full decision coverage, but less than 100% condition coverage and MC/DC. The coverage results for the switching logic *LIO*, *RIO*, *LDL*, *RDL*, had all coverage metrics less then 100%. Clicking on the subsystem hyperlinks in the coverage report, Fig. 6, navigates to the coverage details for that subsystem of the model.

Summary

Model Hierarchy/Complexity:		Total	D1	C1	MCDC
1. Mode Logic SS	98	96%	84%	64%	
2. . . . Mode Logic Chart	97	96%	84%	64%	
3. . . . . SF: Mode Logic Chart	96	96%	84%	64%	
4. . . . . SF: Actuators	72	95%	88%	69%	
5. . . . . SF: LDL	18	92%	88%	63%	
6. . . . . SF: L1	15	89%	88%	63%	
7. . . . . SF: LIO	18	96%	88%	75%	
8. . . . . SF: L1	15	94%	88%	75%	
9. . . . . SF: RDL	18	96%	88%	63%	
10. . . . . SF: L1	15	94%	88%	63%	
11. . . . . SF: RIO	18	96%	88%	75%	
12. . . . . SF: L1	15	94%	88%	75%	
13. . . . . SF: L_switch	12	100%	78%	56%	
14. . . . . SF: R_switch	12	100%	78%	56%	

Figure 6. Model coverage summary.

Figure 7 shows the coverage details for the *L_switch* truth table. Full coverage is not achieved because a number of scenarios were not tested, the coverage analysis has identified that the following cases were not tested:

- For decision 1, [c1&!c3&c4].
- For decision 3, [!c1&c2&c3&!c4] and [c1&c2&c3&!c4].
- For decision 4, [c1&c2]

Similarly, the switching logic for the left IO module could be viewed in the coverage report, but coverage information is also displayed directly on the model through syntax coloring and highlighting. Elements with full coverage are colored green. Elements with no logical content are grayed out.

Figure 8 shows that there are two transitions that have not been fully tested, those gated by [!LDL_act() | RIO_act()] and [!low_press[1] && !L_pos_fail[1]], they are both colored red. Also, the *L1* state is also colored red, further inspection of the coverage report shows that the substate *off* had not been exited when the parent had exited. That is, the left outer actuator did not enter the *isolated* mode from the *off* mode. Similar omissions were noted for the *R_switch* truth table, as well as the switching logic for *LDL*, *RIO*, *RDL*.

13. Function "L_switch"

Parent: mode_logic/Mode Logic SS/Mode Logic Chart

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	0	12
Decision (D1)	NA	100% (12/12) decision outcomes
Condition (C1)	NA	78% (14/18) condition outcomes
MCDC (C1)	NA	56% (5/9) conditions reversed the outcome

Predicate table analysis (missing values are in parentheses)

Hydraulic system 1 Low pressure (Left Outer line)	low_press[1]	T (ok)	T (ok)	F (T)	F (T)	-	-	-
Left Outer actuator position failed	L_pos_fail[1]	-	-	T (ok)	T (ok)	-	-	-
Hydraulic system 2 Low pressure (Inner line)	low_press[2]	F (ok)	-	F (T)	-	T (ok)	-	-
Left Inner actuator position failed	L_pos_fail[2]	F (T)	-	F (ok)	-	-	T (ok)	-
	Actions	2 (ok)	3,5 (ok)	3 (ok)	3,5 (ok)	4 (ok)	5 (ok)	Default

Figure 7. Truth table coverage analysis.

Incomplete coverage means one of three things: (i) there are missing tests, (ii) the design has unreachable or untestable code, or (iii) there are missing requirements (for which tests need to be added). Each of these aspects is considered in determining why full coverage has not been achieved for this design. This paper will now focus on addressing the omissions for the *L_switch* truth table, and *LIO* switching logic, as they identify the changes that will be required for the remainder of the design.

VI. Test and Design Changes

THE coverage of the truth tables and switching logic can be utilized to validate the system and test vector design.

A. Truth Table Logic

First the incomplete coverage for the *L_switch* truth table (Fig. 7) is considered. There were four cases that had not been tested. The first case identified was [c1&!c3&c4]. That is, the case where there are failures in both the hydraulic system 1, and left inner actuator. The second case identified was [!c1&c2&c3&!c4], where there are failures in both the left outer actuator and hydraulic system 2. Both these cases are multiple failure scenarios that would be evaluated true by decision 2 of the truth table. So there are two tests missing.

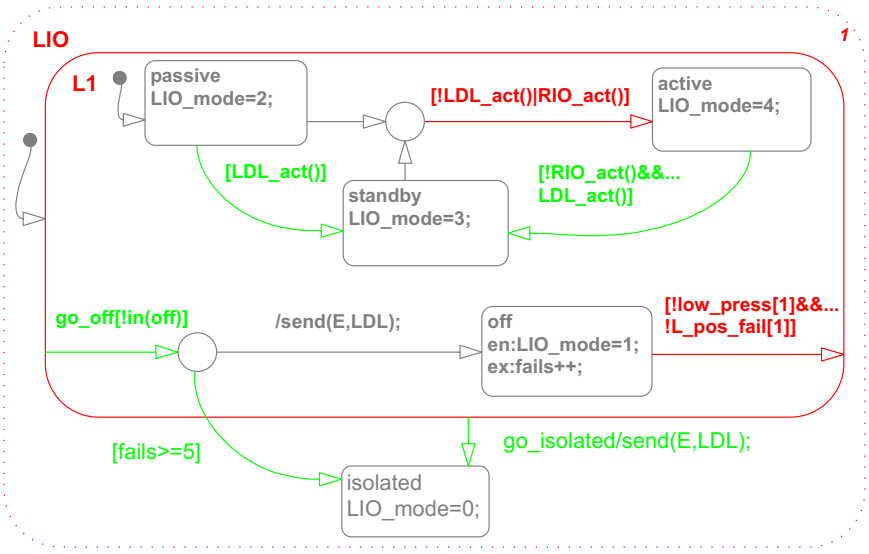


Figure 8. IO module switching logic coverage analysis.

The remaining two cases identified as not being executed are [c1&c2&c3&!c4] in decision 3 and [c1&c2] by decision 4. Now both of these cases can not be evaluated by these decisions as they would have already been evaluated true by decision 2. This highlights the unnecessary specification of condition 1 in both decisions 3 and 4 as false, rather they should be specified as "-" which means the condition is satisfied for either logical value (*don't care*), as decision 2 captures all cases when condition 1 is true. That is, decision 3 can only evaluate to true if condition 1 is false. So the truth table design decisions 3 and 4 can be simplified to be [c2&!c3&!c4] and [c2] respectively.

B. Switching Logic

Upon inspection of the LIO switching logic coverage, the transition to the *active* mode was not fully tested as *RIO_act()* was never true, meaning the left outer actuator had never transitioned into the *active* mode because of the right outer actuator being in the *active* mode.

Also the incomplete coverage of the *L1* state, because of the scenario of going into the *isolated* mode from the *off* mode, had not been tested either. Therefore, two new tests need to be created to check these two conditions. To test the transition into active mode due to the right outer actuator being active, the following test sequence needs to occur:

1. Hydraulic system 3 fails and recovers.
2. The right inner actuator fails.

To test the condition on entering the *isolated* mode from the *off* mode, requires the following sequence to happen:

1. Hydraulic system 1 fails.
2. Left outer actuator fails.
3. Hydraulic pressure 1 recovers, while the left outer actuator is still failed.

Finally, the transition out of the *off* mode, was not fully tested as *L_pos_fail[1]* was never true. Now after further inspection, this is a redundant condition, because if *L_pos_fail[1]* were true, then decision 3 of truth table *L_switch* would evaluate true, and would isolate the left outer actuator. So this means the conditional transition out of the *off* mode can be simplified to be *[!low_press[1]]*, again reducing the complexity of the design.

Similar tests were added and design modifications made to the *R_switch* truth tables and the *LDL*, *RIO*, *RDL* switching logic modules.

VII. Requirements Changes

Consider why the additional tests were originally conceived to be necessary; was it because the requirements were incomplete and ambiguous? To study this further, consider the truth table test that were described in Section VI, specifically the tests for the multiple failures. Looking at the original set of tests, only two of the four multiple failure scenarios were tested:

1. Failures in hydraulic system 1, and hydraulic system 2
2. Failures in left outer actuator, and left inner actuator

The remaining scenarios were not tested:

3. Failures in hydraulic system 1, and left inner actuator.
4. Failures in hydraulic system 2, and left outer actuator.

Both items 1 and 2 were associated with Requirement 2.1.15 given in Table 3. In reading the details of this requirement it becomes clear why items 3 and 4 were not included in the original tests.

Table 3. Multiple failure requirements.

2.1.11 Hydraulic pressure 1 and Left Outer actuator position failures
If a failure is detected in both the hydraulic pressure 1 system and the Left Outer actuator position sensor, while there are no other failures, isolate the fault by switching the Left Outer actuator to the off mode.
2.1.12 Hydraulic pressure 2 and Left Inner actuator position failures
If a failure is detected in both the hydraulic pressure 2 system and the Left Inner actuator position sensor, while there are no other failures, isolate the fault by switching the Left Inner actuator to the off mode.
2.1.15 Multiple failures on Left hydraulics and actuators
If multiple failures are detected in hydraulic pressure 1 or 2 systems and either the Left Outer actuator position sensor or Left Inner actuator position sensor, isolate the fault by switching both the Left Outer actuator and Left Inner Actuator to the isolated mode.

The requirement has been written in a very complicated manner, which is difficult to comprehend, explaining why it was misinterpreted. Close inspection shows that it does not actually state the required behavior correctly as it suggests the following failures:

i Failures in hydraulic system 1, and left outer actuator.
ii Failures in hydraulic system 1, and left inner actuator.
iii Failures in hydraulic system 2, and left outer actuator.
iv Failures in hydraulic system 2, and left inner actuator.

Now item i and item iv are not multiple failures as they relate to the same actuator (either the inner or outer actuator) and have already been captured in Requirement 2.1.11 and Requirement 2.1.12 as shown in Table 3. In addition, item ii and item iii were the missing items, suggesting that the designer read the requirements as: "failure detected in hydraulic pressure system 1 *and* hydraulic pressure system 2 *or* failure detected in the Left Outer actuator position sensor *and* the Left Inner actuator position sensor".

So Requirement 2.1.15 in Table 3 needs to be rewritten to clearly capture all the conditions that it covers. The revised Requirement 2.1.15 is given in Table 4.

Finally, in Table 5, the two requirements that needed to be added to describe the additional switching logic tests are shown.

Table 4. Revised requirement for multiple failures.

2.1.15 Multiple failures on Left hydraulics and actuators
If multiple failures are detected in left hydraulic pressures and actuator positions, isolate the fault by switching both the Left Outer actuator and Left Inner Actuator to the isolated mode. The following combinations trigger this condition: • Failures in hydraulic pressure 1, and hydraulic pressure 2 • Failures in Left Outer actuator position sensor, and Left Inner actuator position sensor • Failures in hydraulic pressure 1, and Left Inner actuator position sensor • Failures in hydraulic pressure 2, and Left Outer actuator position sensor

Table 5. Additional requirements for multiple failures.

2.1.20 Hydraulic Pressure 3 recovers from failure, then Right Inner actuator position fails
If a failure is detected in the hydraulic pressure 3 system and the system then recovers, switch the Right Outer actuator to the standby mode. If a failure is then detected in the Right Inner actuator position sensors, while there are no other failures, isolate the fault by switching the Right Inner actuator to the isolated mode. The Right Outer actuator should move from the standby mode to the active mode.
2.1.21 Hydraulic Pressure 1 fails, Left Outer actuator position fails, Hydraulic Pressure 1 recovers
If a failure is detected in the hydraulic pressure 1 system, while there are no other failures, isolate the fault by switching the Left Outer actuator to the off mode. If a failure is then detected in the Left Outer actuator position sensor as well, while there are no other failures, isolate the fault by keeping the Left Outer actuator in the off mode. If the hydraulic pressure 1 system recovers, and a failure in the Left Outer actuator position sensors is still detected , isolate the fault by switching the Left Outer actuator to the isolated mode.

VIII. The Final Design

By modifying the requirements as discussed in Section VII, and including 11 additional tests and modifying the truth tables and switching logic design as discussed in Section VI, 100% MC/DC coverage on the mode logic model has been obtained. Furthermore, the cyclomatic complexity of the switching logic modules decreased from 18 to 17.

The resulting coverage report of the fault isolation logic truth table of the left actuators is shown in Fig. 9. The report indicates full MC/DC coverage is now obtained for the truth tables as well. Also, by modifying the logic for decision 3 and 4 the cyclomatic complexity of each truth table was reduced from 12 to 10.

For the overall system, the cyclomatic complexity was reduced from 98 to 90. These reductions in complexity indicate that the design has become less complex and easier to interpret.

IX. Conclusions

System design often starts off with a set of high-level requirements (e.g., 'mission requirements') that are gradually refined into a set of detailed requirements from which the subsystem specifications can be derived. Once the subsystem specifications are available, an implementation can be designed. Once the implementation has been realized, extensive testing determines whether the original requirements are met.

In general, requirements are ambiguous, not very rigorous, and even inconsistent. It is desirable to find the problems as early on in the design process as possible. Model-Based Design can be used to that goal by facilitating executable specifications that allow the testing otherwise done after system realization.

Because the executable specification is a computational model of sorts, much information about the internals of a system design is readily available. This is the foundation of verification and validation products that show which parts of a model have been excited in response to one or more test vectors and within what range.

This paper has shown how such *coverage results* can help in making the original requirements consistent and unambiguous, ensuring the design is minimal, and determining the test vectors needed for the given requirements. In the process, a lower complexity of the model design was established.

13. Function "L_switch"

Parent: mode_logic/Mode Logic SS/Mode Logic Chart

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	0	10
Decision (D1)	NA	100% (12/12) decision outcomes
Condition (C1)	NA	100% (12/12) condition outcomes
MCDC (C1)	NA	100% (6/6) conditions reversed the outcome

Predicate table analysis (missing values are in parentheses)

Hydraulic system 1 Low pressure (Left Outer line)	low_press[1]	T (ok)	T (ok)	-	-	-	-	-
Left Outer actuator position failed	L_pos_fail[1]	-	-	T (ok)	T (ok)	-	-	-
Hydraulic system 2 Low pressure (Inner line)	low_press[2]	F (ok)	-	F (ok)	-	T (ok)	-	-
Left Inner actuator position failed	L_pos_fail[2]	F (ok)	-	F (ok)	-	-	T (ok)	-
	Actions	2 (ok)	3,5 (ok)	3 (ok)	3,5 (ok)	4 (ok)	5 (ok)	Default

Figure 9. Modified truth table coverage analysis.

References

[1]Barnard, P., "Graphical Techniques for Aircraft Dynamic Model Development," *AIAA Modeling and Simulation Technologies Conference and Exhibit*, Providence, Rhode Island, Aug. 2004, CD-ROM.

[2]Mai, G. and Schröder, M., "Simulation of a Flight Control Systems' Redundancy Management System using Statemate MAGNUM," 7. User group meeting STATEMATE, April 1999.

[3]Mosterman, P. J., Remelhe, M. A. P., Engell, S., and Otter, M., "Simulation for Analysis of Aircraft Elevator Feedback and Redundancy Control," *Modelling, Analysis, and Design of Hybrid Systems*, edited by S. Engell, G. Frehse, and E. Schnieder, Springer-Verlag, Berlin, 2002, pp. 369–390.

[4]Mosterman, P. J. and Ghidella, J., "Model Reuse for the Training of Fault Scenarios in Aerospace," *Proceedings of the AIAA Modeling and Simulation Technologies Conference*, Providence, RI, Aug. 2004, CD-ROM, ID: 2004-4931.

[5]Seebeck, J., *Modellierung der Redundanzverwaltung von Flugzeugen am Beispiel des ATD durch Petrinetze und Umsetzung der Schaltlogik in C-Code zur Simulationssteuerung*, Diplomarbeit, Arbeitsbereich Flugzeugsystemtechnik, Technische Universität Hamburg-Harburg, 1998.

[6]Stateflow, *Stateflow User's Guide*, The MathWorks, Natick, MA, 2004.

[7]Simulink, *Using Simulink*, The MathWorks, Natick, MA, June 2004.

[8]Simulink Verification and Validation, *Simulink Verification and Validation User's Guide*, The MathWorks, Inc., Natick, MA, 2004.

[9]McCabe, T. J., "A Complexity Measure," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, Dec. 1976, pp. 308–320.