Model Coverage as a Quality Measure and Teaching Tool for Embedded Control System Design

Pieter J. Mosterman, Jason R. Ghidella, and Elisabeth M O'Brien
The MathWorks, 3 Apple Hill Drive, Natick, MA 01760

Abstract - **To systematically establish that a design satisfies its requirements, the design model is analyzed with respect to a set of test cases to establish a measure of so-called model coverage. If less than 100% coverage of model behavior is achieved, the design contains unintended functionality or there may be lacking test cases, which in turn may be because of missing requirements. This paper presents the use of model coverage in education, illustrated by the design of an aircraft attitude control system. Model coverage provides a measure of quality of a design task performed by a student while it can help obtain insight into details of critical behavior of a design and how to correct problems discovered.**

Index Terms – engineering education; Model-Based Design; model coverage; embedded systems

I. Introduction

The use of computational models for Model-Based Design of engineered systems has made modeling an increasingly important aspect of the required skill set of the engineering workforce. Furthermore, verification technologies that ensure that the quality of the design is on par with the requirements are becoming increasingly important. In particular, the growth of embedded systems in devices designed for human interaction renders the design of safety critical code essential.

Teaching the design of engineered systems can be challenging, especially because of the significant inherent creative element. Still, design has to be structured and systematic, for example, to ensure that safety critical systems such as aircraft are designed properly. This has lead to structured approaches that are required by the Federal Aviation Administration (FAA) to certify that the system has been designed with due diligence. In order to be certified by the FAA, airborne software must comply with the DO-178B standard. For the unit testing of safety-critical software, for example, this standard requires the testing process to meet a source code coverage criterion called Modified Condition/Decision Coverage (MC/DC) [1].

More recently, code coverage technology has been adapted to be applied at the model level. The use of coverage in structured design has resulted in the definition of measures of quality of the design. For example, coverage metrics are employed to establish the degree to which functionality is exercised when a test suite is administered. 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.

This paper intends to illustrate how such approaches to structured design can be used in education to provide students with a framework in which to study design and acquire the skills to be successful in devising a design from a set of requirements. Additionally, it is shown how design metrics can be used to evaluate the quality of a design, for example, for the purpose of grading.

It should be noted that, independent of the use of model coverage in the classroom, the understanding and ability to implement model coverage techniques is an important skill for design in industry.

In Section II, design metrics for a structured design approach are introduced. Section III then discusses how these metrics can be used in education. In Section IV an aircraft elevator control system is introduced as an example. In Section V more detailed requirements that can be used for design and that have corresponding tests implemented are presented. In Section VI tests are related to requirements. Section VII then presents some coverage results. Section VIII shows how a test suite may have to be modified, how the coverage results can be used to improve the design and how some coverage results may indicate the need to revise the original requirements. Section IX gives the conclusions of this work.

II. An Introduction to Model Coverage

In the design of software systems, coverage has long been applied to provide a level of confidence that the software operates as desired. In particular, structural coverage in the embedded software development process is necessary to ensure that the requirements-based verification process has not missed important features of the implementation.

In the proceeding, a *condition* is a leaf-level Boolean expression. A *decision* is a Boolean expression that controls the flow of the program, for instance when it is used in an 'if' or 'while' statement. Decisions may be composed of a single condition or several expressions that combine many conditions. Structural testing criteria have been defined that describe the level of coverage of source code.

- **Statement Coverage:** Every statement in the program has been executed at least once.
- **Decision Coverage:** Every point of entry and exit in the program has been invoked at least once, and every

decision in the program has taken all possible outcomes at least once.

- **Condition/Decision Coverage:** Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, and every decision in the program has taken all possible outcomes at least once.
- **Modified Condition/Decision Coverage:** Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken on all possible outcomes at least once, and each condition has been shown to affect that decision outcome independently.

A condition is shown to affect the outcome of a decision independently by varying just that condition while holding fixed all other possible conditions that are part of the decision.

More recently, code coverage metrics have been adapted to block diagrams (e.g., Simulink® [2]) and *statecharts* [3], a state transition diagram formalism with hierarchy, parallelism, and event broadcast. Tools such as Simulink® Verification and Validation [4] have helped this adoption by industry, as part of Model-Based Design, where requirements, tests, and models are tightly coupled [5]. The decision coverage is clearly interpreted as covering decision points in block diagrams such as in switch blocks, whereas in state transition diagrams, they can be interpreted as state activity. The condition coverage in block diagrams relates to blocks with a logical output that is a function of logical input. In state transition diagrams, condition coverage relates to transitions. MC/DC for block diagrams and statecharts is the same refinement of the decision and condition coverage as for code coverage.

Additional coverage measures are provided for models such as block diagrams and statecharts. In particular, truth table coverage captures whether all decisions of a truth table have been evaluated and whether all actions have been invoked. Furthermore, signal range coverage represents the minimum and maximum values assumed by a variable during simulation. Finally, lookup table coverage represents the entries in the table that have been evaluated.

III. MODEL COVERAGE AND EVALUATION

Model coverage technology is providing opportunities to improve design courses as it aids both students and professors in quantifying the quality of a design. In particular it can help students understand the intricate relationships between requirements, design, and testing, while clarifying the importance of verification.

System requirements typically are ambiguous, incomplete, and inconsistent. It is, therefore, standard practice to reengineer requirements as ambiguity is revealed, which is often time-consuming and costly. Model coverage helps pinpoint problems with requirements, such as misinterpreted or ambiguous statements, by highlighting generated tests that are not properly handled by the design. Model coverage combined with traceability will allow students to locate the root cause of test errors. Likewise, coverage analysis will uncover elements in the design that contain superfluous and sometimes duplicate functionality. Adhering to the principle of parsimony,[†] these elements should be removed, thereby reducing design complexity as well as the potential for error. The most straightforward issue that coverage analysis may reveal is the need for additional tests because there is no evidence that a requirement is, indeed, implemented in the design.

The quantitative nature of coverage analysis allows an immediate assessment of an overall system implementation on a scale of 0% to 100%. Depending on how model coverage is used in the classroom, these metrics can be the output the student is graded on. Using coverage analysis as a grading instrument is a convenient measure for evaluation in design courses because it quantifies an otherwise creative process.

For example, students can be provided with a system in an intermediate state of its design and be asked to complete this design. The system may include a partially completed design, requirements, and tests, and the student may be graded on the ability to complete the design while identifying and correcting requirements using model coverage techniques. If the professor has handed the students a system under design with requirements that are intentionally faulty, the student is then graded on the ability to complete the system design according to specifications, set up and run tests, identify the faulty requirements and extraneous elements of the design, then finally modify the design accordingly.

Alternatively, the student could be presented with a design challenge, given a set of requirements, in which the professor could connect to a test harness to determine the quality of the design. Given that the professor is developing the test harness, the requirements must not be ambiguous or incomplete and there must not be any lacking tests. It is then possible for the student to be deducted points for developing a system that contains superfluous elements. Conversely, students could be provided with a completed design, tasking them to determine adequate test cases and grading the task performance based on the percent coverage that is obtained on the submitted test suite.

In this work, an advanced student is given a set of high-level requirements and a preliminary design. From this, the task is to complete the design, re-engineer ambiguous requirements, add missing requirements if needed, and devise a test suite to achieve complete coverage.

IV. DESIGN EXAMPLE

Throughout this work, an aircraft elevator control system [6], [7], [8] will be used to illustrate the concepts discussed.[‡] In other work, it was shown how the design can be combined with test cases for the requirements, specifically, the mode

[†] As per Ockham's Razor, "entia non sunt multiplicanda praeter necessitatem" or "entities should not be multiplied beyond necessity".

[‡] This system, including the model, requirements, and test cases can be obtained upon request.

logic components of the system were considered [45]. Relevant aspects are reiterated here.

A simplified configuration of the redundancy typically found in an aircraft elevator system is shown in Fig. 1. There are two elevators to control the pitch of an aircraft, one on the left and one on the right. Each of the elevators can be controlled by two actuators, only one of which is allowed to be active at any point in time. The four actuators are connected to three separate hydraulic systems. There are two primary flight control units (PFCU) that control either the inner or outer actuators. PFCU1 implements a sophisticated *input-output* (IO) control law that is operational in the nominal mode. A less complicated *direct link* (DL) control law is implemented by PFCU2 to be employed in case of a failure. The IO control then is available for the left (LIO) and right (RIO) actuators. Similarly, the DL control can be applied to the left (LDL) and right (RDL) actuators.

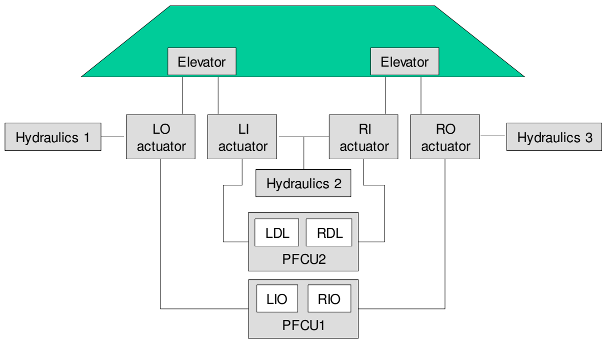


FIGURE 1*
THE ELEVATOR REDUNDANCY CONFIGURATION

The mode logic consists of two components, as shown in Fig. 1. The truth table shown in Fig. 2(a) attempts to isolate the fault from the detected symptoms. 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 entered 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 and D3 are evaluated but their actions are not executed because not all conditions are satisfied. 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. Because a decision is made, no further decisions are evaluated.

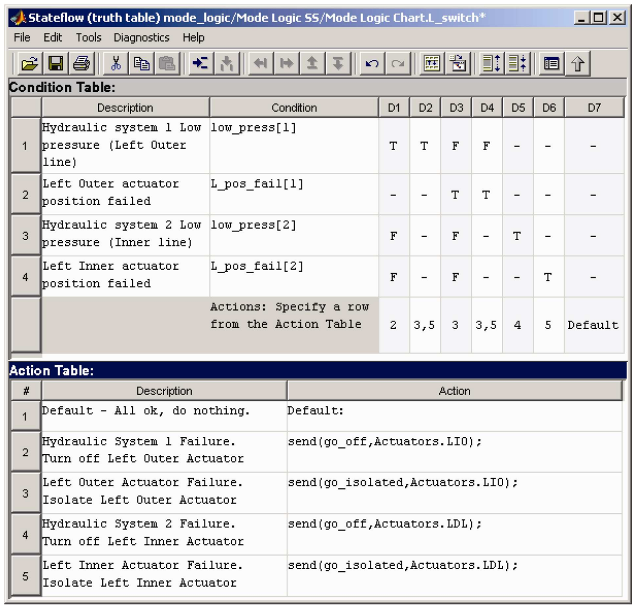
In the "Action Table", the actions 3 and 5 are shown to be

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 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.


Stateflow (truth table) mode_logic/Mode Logic SS/Mode Logic Chart.L_switch*
File Edit Tools Diagnostics Help

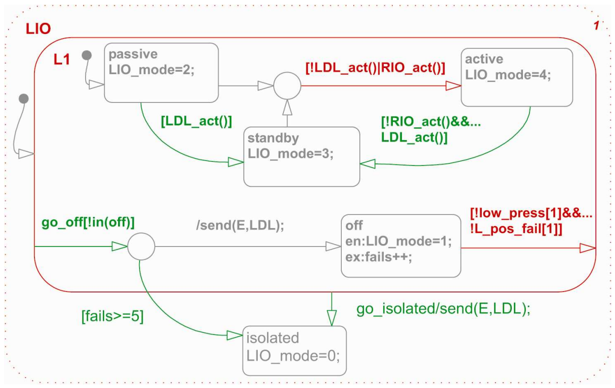
Condition Table:

	Description	Condition	D1	D2	D3	D4	D5	D6	D7
1	Hydraulic system 1 Low pressure (Left Outer line)	low_press[1]	T	T	F	F	-	-	-
2	Left Outer actuator position failed	L_pos_fail[1]	-	-	T	T	-	-	-
3	Hydraulic system 2 Low pressure (Inner line)	low_press[2]	F	-	F	-	T	-	-
4	Left Inner actuator position failed	L_pos_fail[2]	F	-	F	-	-	T	-
		Actions: Specify a row from the Action Table	2	3,5	3	3,5	4	5	Default

Action Table:

#	Description	Action
1	Default - All ok, do nothing.	Default:
2	Hydraulic System 1 Failure. Turn off Left Outer Actuator	send(go_off,Actuators.LIO);
3	Left Outer Actuator Failure. Isolate Left Outer Actuator	send(go_isolated,Actuators.LIO);
4	Hydraulic System 2 Failure. Turn off Left Inner Actuator	send(go_off,Actuators.LDL);
5	Left Inner Actuator Failure. Isolate Left Inner Actuator	send(go_isolated,Actuators.LDL);

(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 as modeled by the hierarchical state transition diagram in Fig. 3(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 may be recoverable or not, respectively.

From Fig. 2(b), it can be seen that the state transition diagram transitions from the *L1* state to the *isolated* state when the *go_isolated* event occurs. As such it is possible to isolate the actuator regardless of the particular state within *L1* it was operating.

Similar state transition diagrams as in Fig. 2(b) for the left outer actuator exist for the other actuators. The logic for each of the actuators is slightly different, though, and the corresponding state transition diagrams are connected via interaction logic.

V. DETAILED REQUIREMENTS

Based on the specifications in Table II, a controller model is designed and the specifications become the requirements of the next design stage [9]. The design then needs to be tested against those requirements to verify its compliance. Additionally, test cases are independently derived from the requirements. These test cases can be executed on the design model to verify that the requirements are met.

TABLE II
SOME 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, white there are no other failures, isolate the fault by switching the Left Inner actuator and the Right Inner actuator to the off 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 for the Passive mode to the active mode. The Left Inner and Right Inner actuators should transition from the Passive mode to the Standby mode.

Producing the detailed requirements and formal specifications tends to be a complex process that may best be reserved for advanced students and the instructor. Alternatively, an existing off-the-shelf design example such as the elevator control system can be employed.

VI. REQUIREMENTS, TESTS, AND ASSERTIONS

To test the requirements, a test harness is established with the test cases created in a *Signal Builder* block and system output is checked via *Assertion* blocks. Using Simulink® Verification and Validation [~~3~~4], an assertion can be associated with a test case in the *Signal Builder* block, so it is enabled when the test is performed. This allows to automatically check for the expected output for a specific test. Additionally, each test case can be associated with a requirement that it is testing (requirements may be provided as elements in, for example, a Microsoft Word document or an Excel spreadsheet). The test harness can be designed and provided in its complete form to the students who design the system according to given requirements. This will test the students' ability to develop a quality design that meets specifications.

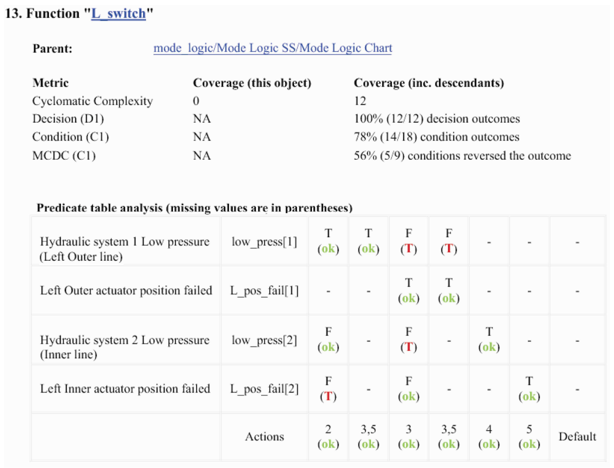
To illustrate, for a test case, labeled "H1 Failure Recover" (see Requirement 2.1.2 in Table II), the hydraulic line must first fail, and then come back online. It is only when the pressure returns to normal that the actuator modes should be checked whether they are in the expected configuration.

In this experiment, a student first created 23 test cases, all of which were associated with verification blocks that checked for expected outputs of the design and requirements for which the tests were created.

VII. COVERAGE

Creating tests based on the requirements and executing those tests is not sufficient to guarantee the design has been fully tested: (i) there may be requirements that are missing tests, (ii) the design may have unreachable or untestable elements, or (iii) there may be missing requirements (for which tests need to be added). Executing each of the 23 tests yielded a coverage report that reveals that even though the student had created tests for all the detailed requirements, full coverage had not been achieved.

Decision coverage is the easiest metric to achieve, where no subsystem of the design achieved less than 89% coverage. On the other hand, condition coverage and modified condition/decision coverage are more restrictive, with some subsystems of the design having coverage as low as 78% and 56% respectively.



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 3[*]
TRUTH TABLE COVERAGE ANALYSIS

The student gained full decision coverage for the truth tables *L_switch* and *R_switch*, but less than 100% condition coverage and MC/DC. The coverage results for the switching logic *LIO, RIO, LDL, RDL*, all had coverage results less than 100%.

Figure 3 shows the coverage details for the *L_switch* truth table. Because a number of scenarios were not tested, the truth table is not fully covered. The coverage analysis has identified that the following cases were not properly 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 coverage report for the LIO switching logic is shown superimposed onto the model in Fig. 2(b). Elements with full coverage are colored green. Elements with no logical content are grayed out. Figure 2(b) 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. The *L1* state is colored red, as well. 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.*

VIII. MODIFICATIONS

This section documents how the student exploited the coverage analysis to improve the system consisting of the requirements, design, and tests. It will be concentrated on the *L_switch* truth table and LIO switching logic.

A. Test Changes

First the incomplete coverage for the *L_switch* truth table (Fig. 3) is considered. The [c1&!c3&c4] case corresponds to failures in both the hydraulic system 1, and left inner actuator. The [!c1&c2&c3&!c4] case corresponds to 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 that were easily added.

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. To test this transition, the following test sequence needs to occur:

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

B. Design Changes

The two other cases identified as not being executed for the *L_switch* truth table (Fig. 3) are [c1&c2&c3&!c4] by decision 3 and [c1&c2] by decision 4. The decisions cannot evaluate these cases as they would have already been evaluated true by decision 2. This highlights the redundant 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.

TABLE III
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 the Left Inner actuator position sensor, isolate the fault by switching both the Left Outer actuator and the Left Inner Actuator to the isolate mode.

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

C. Requirements Changes

Consider why the student had originally omitted the lacking tests. For the multiple failures captured by the truth table only two of the four multiple failure scenarios were tested with the original test suite:

1. Failures in hydraulic system 1, and hydraulic system
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 III. In reading the details of this requirement it becomes clear why items 3 and 4 were not included in the original tests.

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 III. 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".

TABLE IV
REVISED REQUIREMENTS FOR MULTIPLE FAILURES

2.1.15 Multiple failures on Left hydraulic and actuators
If multiple failures are detected in left hydraulic pressure and actuator positions, isolate the fault by switching both the Left Outer actuator and the 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 V
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 isolate 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 isolate mode.

So, Requirement 2.1.15 in Table III needs to be rewritten to clearly capture all the conditions that it covers. The revised Requirement 2.1.15 is given in Table IV. Finally, in Table V, the two requirements that needed to be added to describe the additional switching logic tests are shown. This highlights the importance of requirements engineering, to ensure consistency and unambiguity.

By modifying the requirements as discussed in Section VIII-C, and including 11 additional tests and modifying the truth tables and switching logic design as discussed in Section VIII-A, the student achieved 100% MC/DC coverage on the mode logic model.

IX. CONCLUSIONS

Model coverage is an adaptation of code coverage technology that includes statement coverage, decision coverage, and modified condition/decision coverage (MC/DC). This paper has discussed the use of model coverage in engineering education as an element of a design course. It has highlighted a number of opportunities to employ coverage technology:

- Coverage metrics provide a quantitative assessment of a design.
- Coverage to help students:
  - correct a requirement,
  - correct a design, and
  - systematically test a design.

Furthermore, once complete MC/DC coverage is achieved, the *cyclomatic complexity* of a design can be used as a quantitative measure of the quality of a design. The cyclomatic complexity is an estimate of the McCabe complexity measure [10] for code generated from the model. In a sense, it is a measure of nested decision points in an algorithm and the higher this complexity, the more likely errors are made in the design. Therefore, a lower cyclomatic complexity is an indication of a better design.

The notions are illustrated by discussing the design of a simplified primary attitude control system used in civil aircraft.

REFERENCES

[1] Dupuy and N. Leveson, "An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software," in *Proceedings of the Digital Aviation Systems Conference*, Philadelphia, PA, Oct. 2000.

[2] Simulink, *Using Simulink*, The MathWorks, Inc., Natick, MA, 2004.

[3] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, pp. 231–274, 1987.

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

[5] J. R. Ghidella and P. J. Mosterman, "Requirements-based testing in aircraft control design," in *Proceedings of the AIAA Modeling and Simulation Technologies Conference and Exhibit 2005,* San Francisco, CA, Aug. 2005, CD-ROM, ID: 2005-5886.

[6] G. Mai and M. Schröder, "Simulation of a flight control systems' redundancy management system using statemate MAGNUM," 7. User group meeting STATEMATE, Aschheim, Germany, Apr. 1999.

[7] P. J. Mosterman, M. A. P. Remelhe, S. Engell, and M. Otter, "Simulation for analysis of aircraft elevator feedback and redundancy control," in *Modeling, Analysis, and Design of Hybrid Systems*, S. Engell, G. Frehse, and E. Schnieder, Eds. Berlin: Springer-Verlag, 2002, pp. 369–390.

[8] J. Seebeck, "*Modellierung der Redundanzverwaltung von Flugzeugen am Beispiel des ATD durch Petrinetze und Um- setzung der Schaltlogik in C-Code zur Simulationssteuerung,*" Diplomarbeit, Arbeitsbereich Flugzeugsystemtechnik, Technische Universit¨at Hamburg-Harburg, 1998.

[9] D. J. Hatley and I. Pirbhai, *Strategies for Real-Time Systems Specification*. New York, New York: Dorset House Publishing Co., 1988.

[10] T. J. McCabe, "A complexity measure," IEEE Transactions of Software Engineering, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.