

USING MODEL COVERAGE ANALYSIS TO IMPROVE THE CONTROLS DEVELOPMENT PROCESS

William Aldrich
The MathWorks, Inc.
3 Apple Hill Dr.
Natick, MA 01760

baldrich@mathworks.com

1. Abstract

Graphical simulation tools improve the effectiveness and efficiency of embedded controls development by providing an abstract view of a component or system in a visual paradigm. These modeling tools have unambiguous semantics and can execute designs. Coverage analysis can indicate the completeness and consistency of a set of requirements. Recent advances in code synthesis allow very efficient code to be produced directly from the graphical models. When generated code is used without modification model-based coverage analysis is more effective than the same analysis performed on generated code. Coverage analysis within a modeling tool should be consistent with the executable semantics of the tool and should be presented in a form that is easily associated with the graphical diagrams.

2. Introduction

The context of this paper is the development process used to create large control systems of software and hardware such as those found in aerospace and automotive applications. The typical design process for a large system consists of a sequence of design and implementation steps that rely on documents produced at the output of a preceding step. This process is frequently depicted on a V shaped diagram to indicate the narrowing scope of the successive design steps followed by the increasingly wider scope of integration and testing steps, as shown in Figure 1.

This process requires a considerable effort before any results can be verified. As documented in ¹, the goals of software testing, starting at the lower right of the V diagram, include verifying the consistency of system and high level requirements. When high level requirements change there can be a considerable change in the underlying software. Without techniques to test requirements early in the design process errors propagate from one step to the next and become costly to fix.

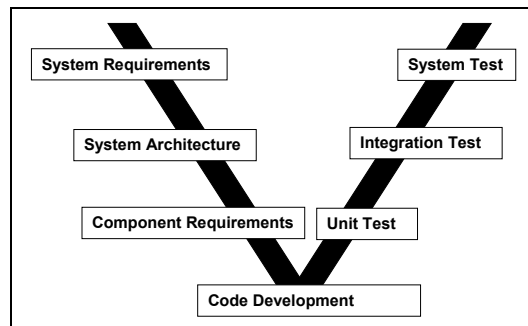


Figure 1: A "v" diagram of system development. The shape indicates the increasingly narrow scope of the requirement and design phase followed by the increasingly wider scope of the integration and testing phases.

A variety of software and hardware tools aid these design steps. Requirements capture and traceability tools are designed to improve the accuracy of requirements and reduce the involved effort when they change. During software testing, code coverage tools provide a measure of test completeness. Finally, logic analyzers are used to unobtrusively capture the software behavior on the final target. This paper focuses on graphical modeling tools used in the heart of the design process and demonstrates how the roles of these tools are enhanced with integrated coverage analysis.

The rest of the paper is organized as follows. Section 3 describes block diagram and state diagram model representations. Section 4 describes how these models are used in the system development process. Section 5 introduces several types of code coverage that serve as the basis for model coverage. Section 6 describes exactly how coverage is implemented in a modeling tool and discusses some special considerations that can result. Section 7 demonstrates the ways that coverage analysis extends modeling tools and aids the development process. Finally, Section 8 investigates the relationship between code coverage

and model coverage. This is particularly significant when code is automatically generated from the model.

3. Graphical Models

This section discusses two fundamental graphical modeling paradigms: block diagrams and state diagrams. Both of these paradigms are familiar to many programmers who have never used modeling tools since they are often used as a means of documentation and specification.

The block diagram examples in this paper were created in Simulink®, a block diagram tool for dynamic simulation. The state diagrams were created with Stateflow®, a state diagram tool that works with Simulink.

A diagram by itself has very limited value if its interpretation is not precise. In addition to diagrams, a useful modeling tool must have a set of semantics that precisely and uniquely determine how the diagram is interpreted and it must have the ability to execute the diagram to observe behavior. The models described in this paper are all executable and deterministic.

3.1. Block Diagrams

Block diagrams emphasize the flow of data between units of computation. The lines on a block diagram are signals that represent data. Each line shows the direction from a source block that assigns values to one or more destination blocks that read those values. Visually, a path through a set of signals and blocks indicates a sequence of computations and intermediate results. A representative block diagram for a P.I. controller is shown below in Figure 2.

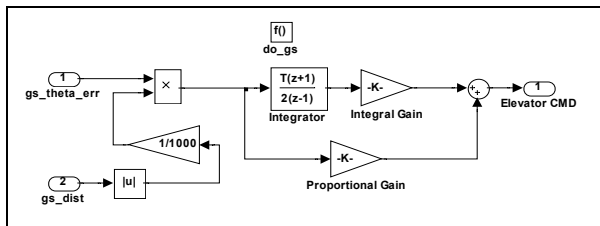


Figure 2: A block diagram of a P.I. controller. This diagram shows how a set of input values are processed to compute an output.

The functional behavior of a block diagram is determined by the set of blocks that compose the diagram, the connections between those blocks, and the underlying semantics of the tool that determine the frequency and ordering of each block execution. The basic blocks available in the diagram should have clear and unambiguous functionality. Ideally, common block patterns should be encapsulated into a single block so a

user can create elaborate functionality with a minimum of blocks. The key to a useful set of blocks is that each one should have an easily identifiable behavior based on its appearance. Some basic Simulink blocks are shown in Figure 3.

To handle scalability, most block diagram tools allow groups of blocks and connections to be organized into a component that is represented as a single block, called a subsystem. This hierarchical representation improves clarity of a design by abstracting details at different levels.

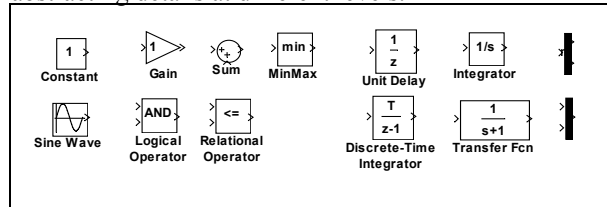


Figure 3: Examples of atomic blocks used to create diagrams.

A practical block diagram tool that can support sophisticated designs needs to provide control flow constructs. Examples of Simulink control flow constructs are shown below in Figure 4. In the first two rows are examples of explicit control flow where the execution of a subsystem causes all the contained blocks to execute. In the bottom row are examples of implicit control flow internal to each block.

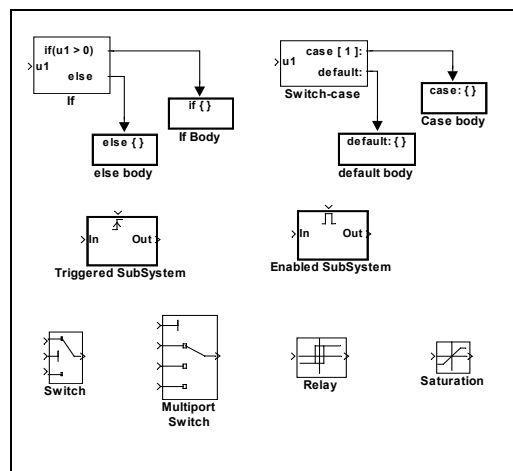


Figure 4: Examples of blocks and constructs that incorporate control flow.

3.2. State Diagrams

State diagrams emphasize the logical behavior of a system. Traditionally, state diagrams have been used to explain how a system with a finite set of modes, or states, can change from one mode or state to another. In Figure 5 rectangles with rounded corners represent

the states in a system. The directed lines from one state to another are called transitions. These indicate the ability to change from one state to another. Transitions are usually labeled with the conditions that must be satisfied before the transition can be taken. Several transitions can originate or terminate on the same state.

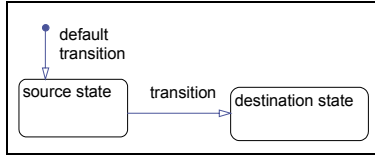


Figure 5: The basic elements of a state diagram.

State diagrams are useful for visualizing logical paths through a series of states. A state diagram can help to clarify the exact sequence of logic that is needed to change from one state to another, particularly when each state has a small number of transitions that originate or terminate on it. Actions associated with states and transitions enable the state diagram to interact with its external environment.

As a design tool, classic state diagrams are limited by scalability problems. Extended state diagrams, like those supported in Stateflow, overcome these limitations with constructs that handle hierarchy, parallelism, and transition re-use. Hierarchy allows states to be grouped together into a superstate so that common transitions only need to be drawn once. Parallelism allows the diagram to be partitioned into several parallel states, each with its own hierarchy of active substate(s). Parallelism prevents the state explosion that results when independent modes or attributes have numerous possible combinations.

A portion of an extended state diagram for an autopilot is shown in Figure 6. Hierarchy allows the states that represent the active modes of the controller to be grouped together in a natural manner. Parallelism allows this logic to be combined in the same diagram with the contents of another parallel state.

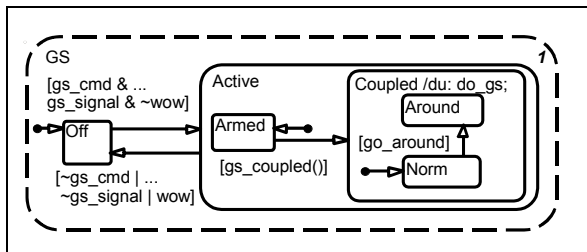


Figure 6: A state diagram showing the logic for enabling ILS glide-slope control.

4. Using Models within a Design Process

Models are used in a design process as executable specifications for the system being developed. Through simulation the model is used to predict behavior. Simulation is particularly useful when the observed behavior is the result of subtle interactions between software and hardware, as in control systems.

An executable specification is used to verify that a design will meet its requirements. These models can identify inconsistent requirements that occur when the model cannot satisfy two or more requirements simultaneously. Once the requirements are verified, the model can be used as a way to predict correct component outputs in the source code.

Useful graphical paradigms are a more natural medium for describing the design and are inherently easier to produce than textual languages. Development time is also reduced when the model is a simplification of the final system. Effective model-based specifications often ignore diagnostics, error handling, calibration support, process scheduling, and other target specific considerations.

Recent advances in code synthesis enable very efficient code to be produced directly from graphical models. When automatically generated code is used without modification the graphical model serves as an implementation. The graphical environment can be thought of as a language equivalent to any other programming language.

Using models as implementations is motivated by the advantages of graphical representations over source code. Rapidly generated specifications can be used as a starting point for developing model-based implementations. The process of successive model refinements and eventual automatic code generation represents a complete model-based development process.

5. Code Coverage Analysis

Coverage analysis is used to dynamically analyze the way that a program executes. It provides a measure of the completeness of testing based on the code structure, known as white box testing. A code fragment, shown in Figure 7 below, will clarify the meaning of each coverage metric.

The simplest form of coverage is called *statement coverage*, sometimes abbreviated as C1. Full statement coverage indicates that every statement in a program has executed at least once. The limitation of statement coverage is that it does not completely analyze the control flow constructs within a program. For example, when an if statement does not have a matching else, full coverage only requires that the if

evaluate to true. In the example a single test case can achieve complete statement coverage. By setting $x=15$ and $y=2$ every statement executes.

```
x = 2* a;
z = 3;
if (x>5 & y<4) {
    z = 10;
}
out = 1;
if (y*x==30) {
    out = 4;
}
```

Figure 7: A code fragment to illustrate coverage metrics.

A more rigorous form of coverage is *decision coverage*, sometimes abbreviated as C2. Full decision coverage indicates that each control flow point in a program has taken every possible outcome at least once. In a well-structured program, such as one written in a higher order programming language, decision coverage implies statement coverage³. In the example there are two decisions, $x>5 \ \& \ y<4$, and $y*x==30$. Both of these decisions can be either true or false, so at least two test cases are needed for full decision coverage. One case sets both to true. The other sets both to false.

Decision coverage provides a good measure of completeness but it ignores the complications that result when a decision is determined by a logical expression containing the logical operators AND or OR. In this case, the Boolean inputs to logical expressions are analyzed using *condition coverage*. Full condition coverage indicates that every input to a logical expression, called a condition, has taken a true and a false value at least once. In the example there are three conditions, $x>5$, $y<4$ and $y*x==30$. Complete coverage can be achieved with two test cases by testing all the conditions as true and then all the conditions as false.

MC/DC coverage, or modified condition-decision coverage, provides an even more rigorous analysis of conditions. Full MC/DC coverage implies that each input to a logical expression has been shown to independently change the expression outcome while the other conditions are held constant. In the above example the line `if (x>5 & y<4)` will require three tests. One test will set both conditions as true, one test will just set $x>5$ as true and the last test will just set $y<4$ as true.

6. Model based interpretations of coverage

The basic goal of model based coverage is to provide the equivalent information of code coverage in

the context of a graphical model under simulation. While statement coverage is difficult to interpret in any context other than textual languages, the other structural metrics can be interpreted based on the underlying model semantics.

The challenge is that graphical tools have many more control flow constructs than a programming language. A further complication is that the control flow related to a model may not be immediately obvious. A model design might be simulated with an engine that includes an extensive amount of logic for interpreting and analyzing the model and trapping internal programming errors. It is important to omit everything but the coverage that relates to the design dynamics.

Consider a multiport switch block that chooses between several inputs based on the value of a control signal, as shown in Figure 8. The number of inputs can be specified when the block is instantiated. Whenever this block is used in a model it is equivalent to a switch-case construct having as many outcomes as the block has inputs. This is treated as a single decision with n possible outcomes for purposes of decision coverage. A coverage tool simply needs to record what cases execute. Each outcome can be recorded with a count of the number of times it was executed or with a flag to indicate when it was executed at least once.

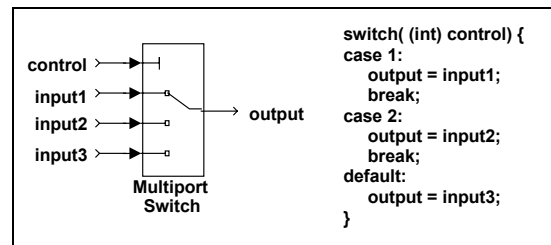


Figure 8: A multiport switch with three selection inputs and its equivalent implementation in C code.

When creating a coverage tool, each type of block or basic object within the tool must be analyzed to determine if it can contribute to the control flow within a design. Procedures must be developed for each of these block types so that the correct data structures are initialized for each instance in a design before the start of a simulation. Another set of procedures must be produced to dynamically update the coverage data when the instantiated object is executed.

Sometimes a model construct does not have a unique code implementation so that the equivalent coverage is not well defined. A good example is a min or max construct with more than two inputs. Figure 9 shows a min block with three possible C code implementations, each of which might have different coverage for the same test cases.

This author proposes that the best way to handle model constructs that don't have well defined model coverage is to choose a coverage requirement that will guarantee full coverage in all of the likely implementations. By requiring that each input is less than the others for at least one occurrence, all three of these Min block implementations will be fully covered. In fact, any implementation that compares the inputs deterministically will be fully covered with this requirement.

6.1. State Diagram Considerations

Most of the state diagram logic is determined by transitions so they are the central elements within a coverage report. The way that states and transitions are analyzed depends on the tool semantics. In some tools a transition must always execute when the diagram is updated and self-transitions are required to explicitly indicate that a state will remain active after an update. A tool might force the transitions from a common state to be mutually exclusive or might use an ordering to resolve conflicts

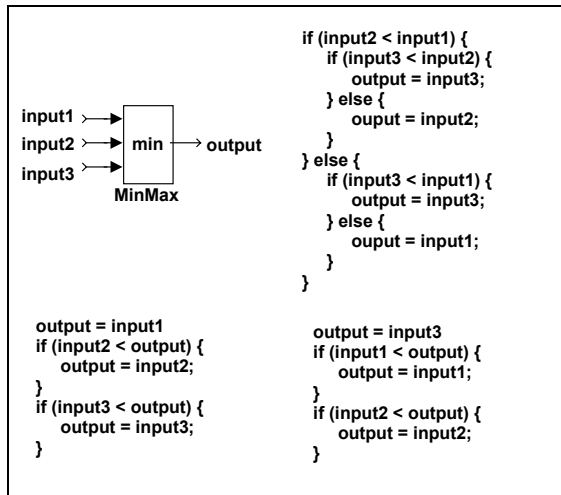


Figure 9: A min block with three C code implementations.

In Stateflow, transitions are ordered based on their label structure and unconditional transitions are executed by default after every other transition is tested false. With this semantic, every conditional transition has a definite point when it is tested and either a true or false outcome. Full coverage requires that every conditional transition be tested at least once as false and at least once as true. Unconditional transitions have no logic because whenever the diagram processing reaches the point where that transition would be tested it is known a priori that the transition will be taken.

6.2. Context Dependent Logic

Within a modeling environment control flow may be the result of interactions between objects instead of being isolated to a single object. This is especially true in extended state diagrams because the logic for traversing a hierarchy is combined with the transition logic. An example is shown in Figure 10. When a superstate is the source of a transition, as shown on the bottom, it must contain logic to determine which of its children should be exited. If a superstate is not the source of a transition, as shown on the top, that logic is unnecessary. A model coverage tool will sometimes need to look at related objects to determine what type of control flow exists.

It is important to distinguish analyzing context from other types of static analysis, like reachability. It is quite simple to create code or models that have coverage paths that cannot be exercised. The role of a coverage tool is to determine what portions of an implementation have executed. Even if some elements cannot be reached they should still be reported as uncovered.

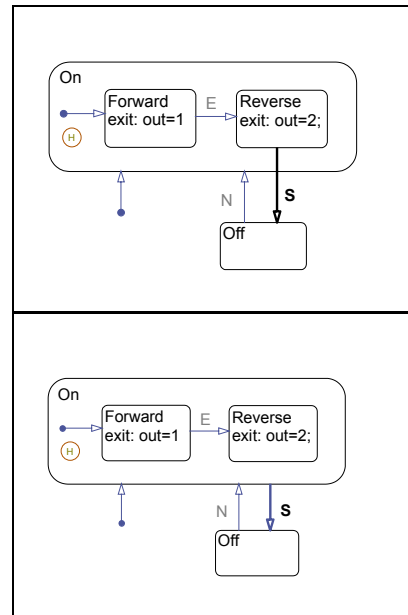


Figure 10: An example of state diagram logic dependent on context. On the top the transition labeled S explicitly exits the state Reverse. On the bottom the transition labeled S will implicitly cause either the Forward or Reverse state to exit requiring that logic is added to determine which exit action will execute.

7. Uses of Model Coverage

The uses of model coverage are similar to those of code coverage and depend on where in the

design process the tool is being used. When applied to a specification, model coverage can be used to indicate the completeness of a set of requirements. Normally this type of information, which is useful for identifying gaps in the specification where the designer and implementers may have different ideas of the correct behavior, is not available until the code is produced.

7.1. Measuring Requirements Completeness

In a typical development process the target software is based on written requirements. The functional requirements, which deal with input/output behavior, are verified by executing test cases. Consistent and complete requirements are critical to a successful development process.

To demonstrate how model coverage aids this process we will consider a hypothetical autopilot design. The goal of the design is to control altitude of a plane according to set points for altitude and vertical speed and to control the landing approach based on ILS glide-slope instruments.

We assume that the system architecture dictates the following inputs and outputs:

Inputs

1. Actual plane altitude
2. Boolean autopilot enable flag
3. Desired altitude
4. Desired vertical speed (climb and fall)
5. Boolean update flag to read new set points
6. Boolean glide-slope enable flag
7. Boolean landing go-around flag
8. Boolean flag indicating GS signal
9. Glide-slope angular error
10. Distance to glide-slope antenna
11. Boolean weight on wheels flag

Outputs

1. Commanded elevator position
2. Flag indicating GS control is armed
3. Flag indicating GS control is coupled.

We will restrict our attention to the control logic contained in the state machine. The state machine uses the same set of inputs but instead of outputting an elevator command it sends one of three trigger commands to invoke either the altitude hold controller, the altitude climb controller, or the glide-slope controller.

We consider a simplified list of state machine requirements that is deliberately incomplete:

1. Altitude climb control is entered whenever $|\text{altitude}-\text{desired}| > 1500$

2. Altitude hold mode is entered when $|\text{altitude}-\text{desired}| < 30 * \text{ts} * (\text{alt_rate} / 60)$
3. Altitude hold control and altitude climb control are disabled when autopilot enable is false
4. Glide-slope control is armed when its enable flag and signal flag are both true.
5. Glide-slope control becomes coupled when the control is armed, angle error > 0 , distance < 10000

An executable specification for the controller is implemented using Simulink and Stateflow. The top level diagram is shown in Figure 11. There is very little processing in the Simulink diagram other than latching the controller set points.

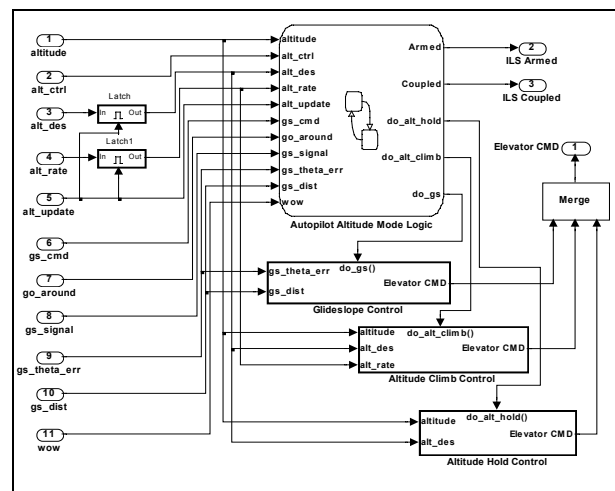


Figure 11: The Simulink block diagram of the autopilot controller.

The contents of the state diagram are shown in Figure 12. Initially glide slope control is off and the altitude control is disabled. In both of these states a true value for the weight on wheels flag will prevent the control from being activated and will disable it when it is active. Glide-slope control transitions from the off mode to the armed mode according to requirement 4 and to the coupled mode according to requirement 5. A go-around mode is used to represent when a landing approach has been aborted. The block at the bottom of the figure is a graphical function that calculates when the criteria for glide-slope coupling is true. The function is invoked in the transition from armed to coupled.

The altitude state in the middle of the figure determines when the hold controller and climb controller are triggered based on Hold or Climb_rate being the active state. The transition between these two states is dictated by requirements 2 and 3. An

additional graphical function is used to encapsulate the logic.

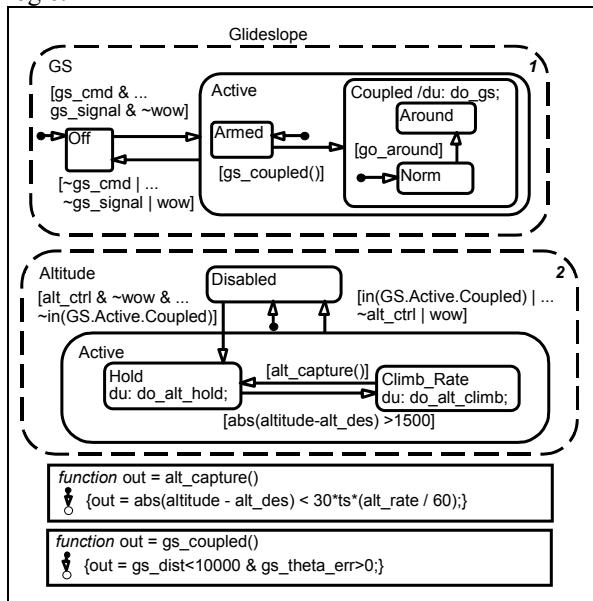


Figure 12: The state diagram for the autopilot. The top state contains logic for the glide-slope control and the bottom state contains logic for the altitude control modes.

The design is verified with test cases derived from the requirements. Usually independence is maintained between the design implementation and the derivation of test cases to reduce the possibility of a common misinterpretation.

The table below describes the set of test cases derived from the list of requirements.

1	Set altitude==desired and update flag true, increase desired altitude by 2000 and verify climb mode is triggered.
2	Start with altitude climb control active and altitude==desired-2000. Gradually increase altitude and verify that when $ altitude-desired < 30*ts*(alt_rate / 60)$ is true altitude hold mode becomes triggered
3	Set altitude==desired and change enable flag from true to false and verify that control triggering stops Set altitude==desired-2000 and change enable flag from true to false and verify that control triggering stops
4	Set glide-slope enable true and transition the signal flag from false to true while angle error is < 0 , verify the armed flag becomes true.
5	Start with glide-slope armed and distance <10000 increase the angle error to >0 and verify the coupled flag becomes true

Each of these test cases is executed within the model by running a simulation and ensuring the verification requirement is satisfied. The model coverage tool that was implemented for Simulink and Stateflow is capable of aggregating coverage results from several tests into a single integrated report. This report indicates how complete the requirements based test cases are in terms of the model structure.

A portion from the top of the report is shown in Figure 13. Overall, the test cases executed 78% of the decision outcomes, 75% of the condition values and demonstrated MCDC for 50% of the conditions. Observing coverage relative to model hierarchy can indicate specific portions of the diagram that require more testing. In this case the states related to glide-slope control are not as fully covered as the altitude control states.

Model Hierarchy/Complexity:	Test 1		
	D1	C1	MCDC
1. req_test	35 84%	70%	50%
2. Logic	25 78%	75%	50%
3. SF: Logic	24 78%	75%	50%
4. SF: Altitude	11 100%	83%	67%
5. SF: Active	4 100%	NA	NA
6. SF: GS	13 61%	67%	33%
7. SF: Active	6 50%	NA	NA
8. SF: Coupled	3 33%	NA	NA

Figure 13: A portion of the HTML coverage report indicating the coverage for hierarchical portions of the Stateflow model.

More detailed coverage information is shown in the report portion in Figure 14. The transition detailed in this figure was never tested true. The hyperlinks within this report, which are underlined, navigate the user to the related object in the graphical model.

Table 1 summarizes all of the missing coverage items. As expected most of the items were deliberately omitted from the requirements, such as the weight on wheel flag and the go around abort landing flag. One exception is the transition to disable glide-slope control that was never taken. In this particular case we would probably want to add a requirement that describes how the glide-slope controller can be disabled.

In some cases, even if the requirements are complete we might not have derived sufficiently detailed test cases. The manual process of resolving missing coverage helps ensure that each item within the model is related to a requirement and that each requirement has one or more derived test cases.

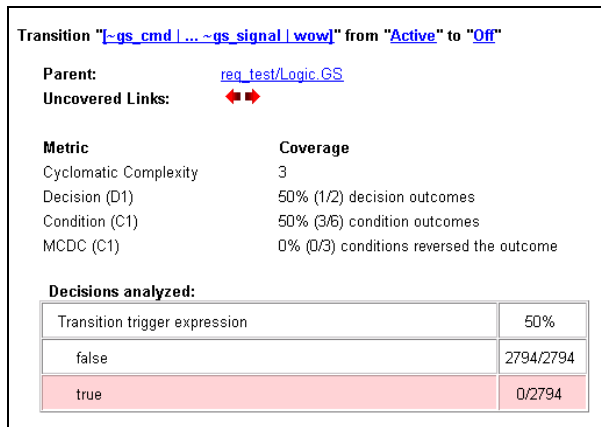


Figure 14: The detailed coverage entry for the transition from GS.Active to GS.Off shown in Figure 12. Every time the transition was tested it was false.

7.2. Coverage Analysis for Model Based Implementations

When the graphical designs from a modeling tool are used as implementations they should be treated as the most detailed component design in the development process. This means that model coverage tools must support the detailed verification steps imposed on the design process. Model coverage can be used to simplify unit test development, to complement the code coverage information, and to aid in demonstrating the equivalence of a model and its generated target.

7.2.1. Test Generation

Coverage analysis is used as an aid to test generation by capturing the test deficiencies and using that information to develop new tests. While the deficiencies can be captured with either model

coverage or code coverage, it will be easier to determine the design significance from the model and identify the expected results just as source code coverage would be easier to interpret than machine code coverage.

When source code is automatically produced from a graphical design the coverage analysis of the resulting code has the same limitations as analyzing coverage of machine instructions. Generated code is an interpretation of model behavior within the semantics of a textual language and the design significance of a particular statement may not be obvious. A good example of the need for model coverage is shown in Figure 15. In this example, a combinatorial logic block is used to select a desired output. The coverage report clearly indicates the four possible ways the block can execute and associates a simple input requirement for each case. Contrast this information with the code fragment. The way the block executes is determined by a pair of computations on an intermediate variable and an array de-reference. It would be much harder to relate these code statements with the basic block behavior.

In unit testing, requirement-based test cases frequently need to be augmented with additional cases to achieve some mandated coverage. Consider a transition with the guarding condition "[(A & B) | (C & D)]", as shown in Figure 16. You may have tested all the requirements and demonstrated that A, B, C, and D have been tested as true and false but in order to achieve full MC/DC coverage you need to demonstrate that each condition can independently change the outcome of the transition. If the conditions in the transition are intermediate values computed from another set of inputs you might need to analyze the model to determine how to achieve each value. This

Table 1: A summary of the uncovered items after simulating all of the requirement-based test cases

Model Object	7.3. Missing Coverage
Transition from Altitude.Disabled to Altitude.Hold	WOW flag was never true such that it prevented the transition from being taken.
Transition from Altitude.Active to Altitude.Disabled	WOW flag was never true such that it caused the transition to be taken.
Transition from GS.Off to GS.Active	WOW flag was never true such that it prevented the transition from being taken.
Transition from GS.Active to GS.Off	Transition was never true.
State GS.Active	State never exited when either Armed or Coupled substates were active.
State GS.Coupled	Never exited when either Around or Norm substates were active. Never called the Around substate.
Transition "[go around]"	Transition was never true.

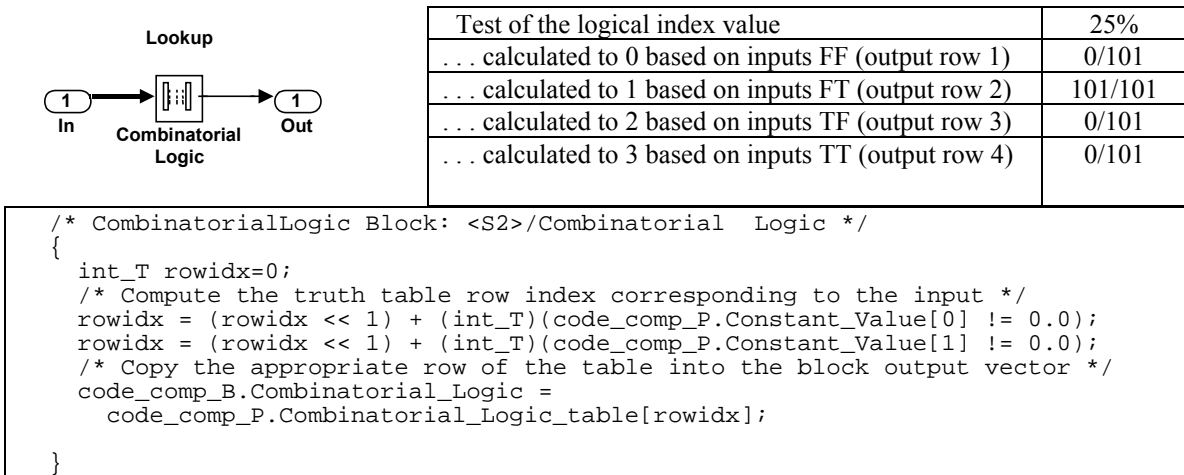


Figure 15: A comparison of the coverage report and the generated code for a single instance of the combinatorial logic block. The generated code would serve as the basis for reporting code coverage from the generated target.

type of analysis is easier when the information is clearly displayed relative to the model, as shown in Figure 16 below.

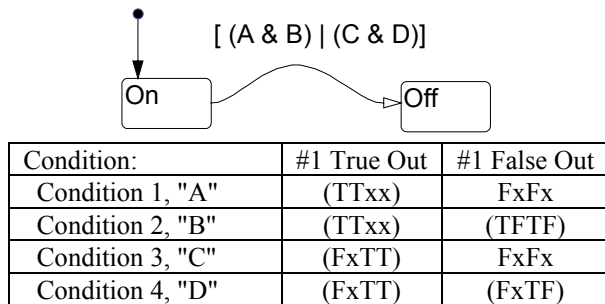


Figure 16: MC/DC coverage information for a transition with four conditions. The coverage information lists the specific combinations to satisfy MC/DC coverage.

7.3.1. Demonstrating Model and Target Equivalence

The ultimate goal of the system development process is to create a properly functioning target. It is not sufficient to simply create a functional model or a source file that behaves correctly in a simulated environment. A model-based approach is to divide this problem into two steps: first show that the model is correct and meets all requirements, then show that the target is equivalent to the model.

Equivalency indicates that the output of the code generator and compiler are correct in the single design instance being tested.

The diagram in Figure 17 shows the process of testing equivalency. At a minimum, equivalency must be demonstrated with test cases that achieve all mandated test coverage. The test coverage can be measured with either model or code coverage. Code coverage has the advantage that it is determined independent of the modeling tool. Model coverage provides an alternative to code coverage that simplifies the creation of needed test cases. A user can iteratively extend their test cases using model coverage until full model coverage is achieved and afterwards confirm that code coverage is complete. Model coverage also provides a test for errors of omission in the target that might not be exercised under full code coverage.

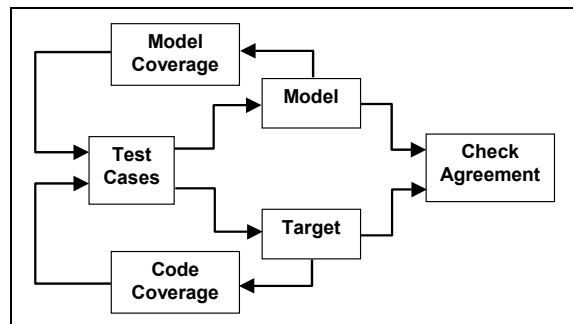


Figure 17: A diagram showing the iterative process of testing model and target equivalency. The top loop and bottom loop should be iterated until full coverage is achieved for the model and target. If the outputs agree in all test cases the goal has been achieved.

The process of testing equivalency is unique in that the expected outputs for each test case do not have to be provided. This makes equivalency testing ideally suited to automation. A tool can randomly or systematically generate input values and test the output agreement until a disagreement is found or full model and code coverage are achieved.

8. Comparing Code Coverage and Model Coverage

This section describes in detail the mechanisms that can cause disagreements between model coverage and code coverage. These are the same mechanisms that cause disagreements between source code coverage and object code coverage. These disagreements may be the result of errors in the compilation process or may be the unintended result of a correct translation that added or removed logical constructs.

Several optimizations designed to reduce the size of the generated code or improve its performance dramatically change the coverage. In order to meaningfully compare coverage results from a model and generated code it is important that the code generator maintain a one-to-one mapping between elements in the model and the corresponding elements in the generated code.

An inline optimization is shown in Figure 18. This is an optimization where the generated code duplicates the contents of a reusable procedure within each of its callers. The optimization is designed to improve performance by eliminating the procedure call at the expense of a larger program. In the model, coverage of the re-usable component is the union of the coverage from each caller. Each call point might only cover a portion of the component while the union is fully covered. In the generated code each context will be treated separately and full coverage will require full coverage from every caller.

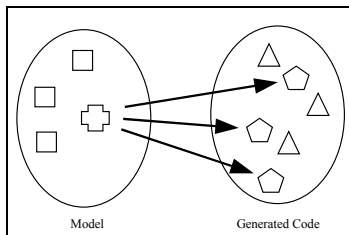


Figure 18: An Inline optimization where a single re-usable component in the model is replaced by a set of distinguishable constructs in the generated code, one for each use in the model.

The opposite of an inline optimization is a re-use optimization, shown in Figure 19. This optimization takes a set of equivalent but distinguishable constructs and implements them in a single reusable component to conserve program size. This optimization will inevitably improve coverage by indicating the compiled coverage is the union of all uses.

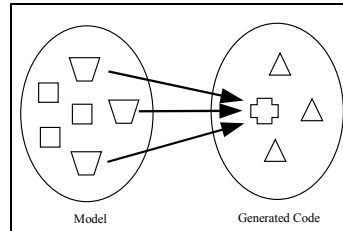


Figure 19: A Re-use optimization that takes a set of equivalent but distinguishable design constructs in an model and replaces them with a single re-usable construct in the generated code.

Another type of optimization that changes coverage is dead code removal, as shown in Figure 20. If a compiler is able to statically identify dead code it might omit that code from the generated code to conserve space. The dead code that was uncovered in the model would be omitted in the generated code, indicating more complete coverage.

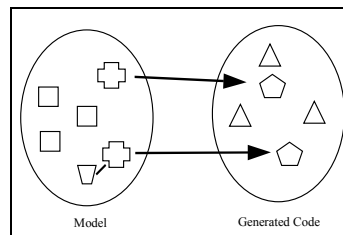


Figure 20: A dead code optimization where part of the model does not affect the generated code because that portion is non-functional.

It is important to realize that there is only a tenuous agreement between coverage points and functional behavior. Even in the same language, small permutations in an implementation that do not change functionality can change the resulting coverage. Even if the code generator does not use in-lining, re-use, or dead code optimizations it might reorganize the logic slightly so that coverage results are different. Consider the code in Figure 21 in which the code on the left has three decisions while the code on the right has a single decision.

The important point is that all coverage is inherently imprecise and should not be treated as an absolute measure. All measures of coverage are

intended to indicate a minimum level of testing and provide some comparative measure between two sets of tests.

<pre> if (C1) { if (C2) { func1(); } else { if (C3) { func1(); } } } </pre>	<pre> if (C1 && (C2 C3)) { func1(); } </pre>
---	---

Figure 21: An example of two functionally identical code fragments having different coverage requirements based on the coded structure.

9. Conclusions

This paper has demonstrated the need for coverage analysis within behavioral design tools. Tool vendors have traditionally focused their efforts on automating the design process and left the testing and verification activities to their users. Coverage analysis is a key component of testing and verification that is particularly difficult to achieve manually or by using an add-on product, but is relatively simple to incorporate within a simulation tool.

The concepts discussed in this paper have been implemented in the Model Coverage tool, sold as part of the Simulink Performance Tools, available for use with Simulink and Stateflow from the MathWorks Inc.

10. References

[1] "Software considerations in airborne systems and equipment certification," Document RTCA/DO-178B, RTCA Inc., December 1992.

[2] "Final report for clarification of DO-178B," Document RTCA/DO-248B, RTCA Inc., October 2001.

[3] Beizer, B., "Software Testing Techniques," (Van Nostrand Reinhold, New York, 1990) Second Edition.

[4] Conrad M., Weber M., and Müller O., "Towards a methodology for the design of hybrid systems in automotive electronics," Proceedings of the International Symposium on Automotive Technology and Automation (ISATA '98), 1998.

[5] Chilenski, J. J. and Miller, S. P., "Applicability of modified condition/decision coverage to software testing," Software Engineering Journal, September, 1994.

[6] Degani A. and Heymann M. "Pilot-Autopilot Interaction: A Formal Perspective" 1999 <http://ic.arc.nasa.gov/publications/pdf/2000-0181.pdf>

[7] Harel, D. "Statecharts: a visual approach to complex systems," Science of Computer Programming, vol 8, pp. 231--274, 1987.

[8] Offutt, A. J., Xiong, Y., Liu, S. "Criteria for Generating Specification-Based Tests," Proceedings of Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'99), Las Vegas, Nevada, USA, October 18--21, 1999, pp. 119--129.

[9] Patel, S., Smith, P., et al., "CACSD in Production Development: An Engine Control Case Study", 2000, Global Powertrain Conference

[10] Poston, R. M., "Automating Specification-Based Software Testing," (IEEE Computer Society Press, Los Alamitos, 1996)

[11] Rauw, M.O., 1993, "A Simulink Environment for Flight Dynamics and Control analysis – Application to the DHC-2 "Beaver"" (MSc-thesis, Delft University of Technology, Faculty of Aerospace Engineering, Delft, The Netherlands, 1993).

[12] Stateflow User's Guide, Version 4.0, The MathWorks Inc., September, 2000.

[13] Using Simulink, Version 4.0, The MathWorks Inc., November, 2000.

[14] <http://www.allstar.fiu.edu/aero/ILS.htm>