

Coverage Analysis for Model Based Design Tools

William Aldrich
The MathWorks, Inc.
3 Apple Hill Dr.
Natick, MA 02478

baldrich@mathworks.com

1 Abstract

Graphical modeling tools that provide an abstract view of a component or system in a visual paradigm improve the effectiveness and efficiency of software development processes. The most useful modeling tools have unambiguous semantics and can execute designs. These models have traditionally been used as behavioral specifications. Coverage analysis within a behavioral specification 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 the graphical models become implementations and integrated coverage analysis becomes a more effective alternative to 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 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.

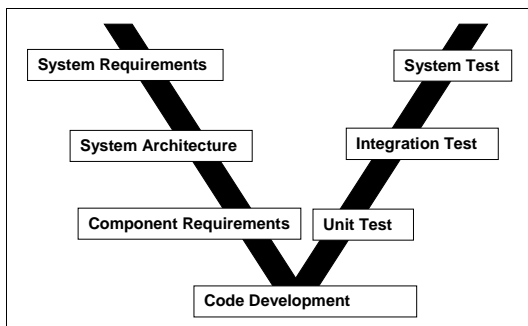


Figure 1: A "v" diagram of system development. The shape indicates the increasingly narrow scope of the requirement and design phase followed by the increasingly wider scope of the integration and testing phases.

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.

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 based 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.

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 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 and represent data flow. Each line shows the direction of flow 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. While the explicit order in a textual representation hides the arbitrariness of certain procedural steps, a block diagram adds clarity to a design by reinforcing the required order of processing. A representative block diagram for a P.I.D controller is shown below in Figure 2.

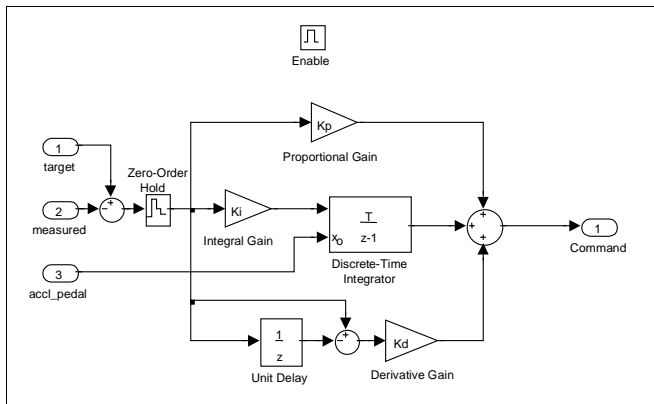


Figure 2: A block diagram of a P.I.D controller. This diagram shows how a set of input values are processed to compute an output.

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. In Simulink, these components are called subsystems and are often represented by a block having labeled input and output ports, such as the block on the extreme right in Figure 3. Subsystems can be nested within each other so that a simple diagram can represent a very elaborate set of atomic blocks.

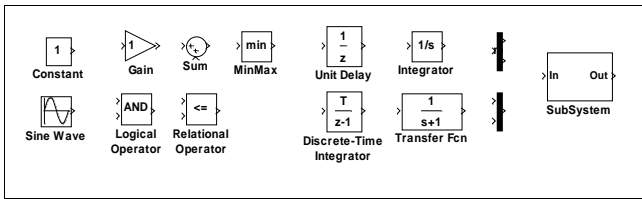


Figure 3: Examples of atomic blocks used to create diagrams.

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. Some control flow may be localized within a particular type of block. This can be thought of as implicit control flow. More powerful constructs control the execution of entire sets of blocks. The blocks in Figure 4 that have signals directed to the top are conditionally executed subsystems. They can be thought of as explicit control flow. The signals are used to control the execution of the blocks underneath.

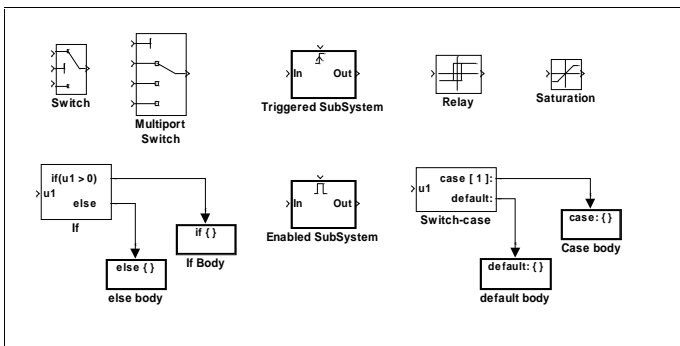


Figure 4: Examples of blocks and constructs that incorporate control flow.

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.

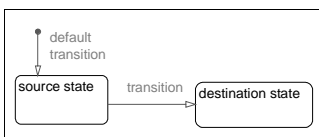


Figure 5: The basic elements of a state diagram.

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 a cruise control application is shown in Figure 6. Hierarchy allows the states that represent the powered-on 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.

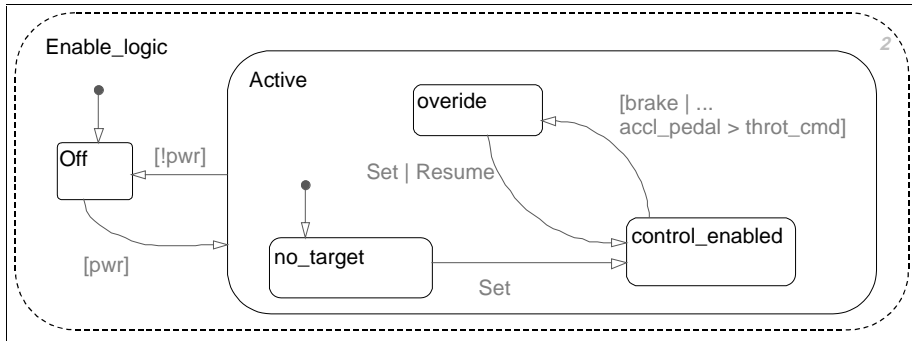


Figure 6: A state diagram showing the logic for a cruise control. When power is enabled, i.e., the condition `[pwr]` is logically true, the active state changes from `Off` to the `no_target` substate of `Active` (`Active.no_target`). When the `Set` event occurs, the mode changes to `Active.control_enabled`.

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.

Effective use of models as specifications requires that they be produced in considerably less time than source code. Useful graphical paradigms are a more natural medium for describing the design and are inherently easier to produce. 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.

When a model is used as a specification it implies that further design effort is needed to produce the final implementation. 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. When the model is used as an implementation there is no further design effort to produce the source code and the graphical environment can be thought of as an implementation language equivalent to any other programming language. This often requires that additional target information be added to the model that might have been omitted in a specification.

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 coverage [2, pg. 74]. 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. Some metrics are difficult to interpret in any context other than textual languages. There really is no equivalent to statement coverage within a graphical tool because a single block may be implemented in several different ways, each with a different number of statements. It is, however, possible to identify the equivalent control flow within a modeling tool and since branch coverage in a well structured program implies statement coverage this is a logical starting point.

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. The

control flow within 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 dynamic implementation of the design. You would never expect a code coverage tool to require that an if block has a certain number of statements, even if this logic was part of the compiler.

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.

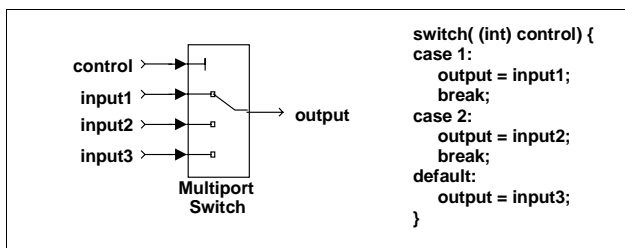


Figure 8: A multiport switch with three selection inputs and its equivalent implementation in C code.

This example shows the fundamental technique to 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. Model coverage could be determined by using the particular implementation within the interpretive engine or code generator, but this would introduce dependencies on intermediate variables that have no design significance and a permutation of the inputs would change the coverage result.

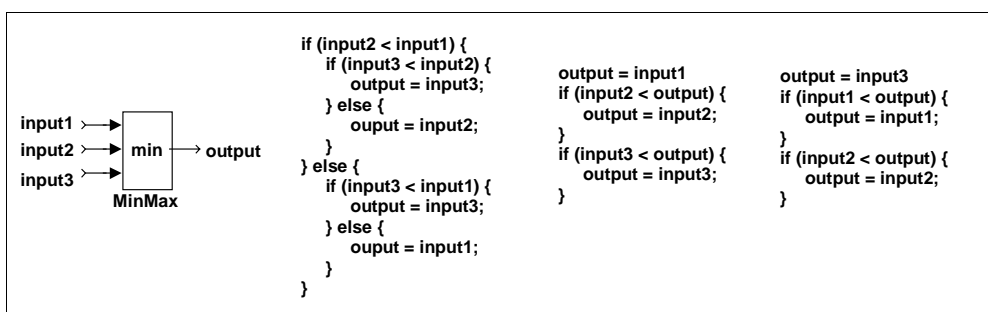


Figure 9: A min block with three C code implementations.

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.

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

The control flow within a model construct is frequently dependent on the context it is used in. This is especially true in extended state diagrams because the logic for traversing a hierarchy and updating a diagram is dependent on the network of transitions that connect the states together. An example is shown in Figure 10. When a superstate is the source of a transition, as shown on the right, 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 left, that logic is unnecessary. A model coverage tool will sometimes need to look at related objects to determine what type of control flow exists.

Context is usually not an issue for code coverage. An if statement always has control flow, regardless of where it is used. 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 which can not be exercised. Most code coverage tools do not use context to rule out unachievable paths. The user will simply find that these paths never achieve coverage and will hopefully recognize the design flaw. In fact, coverage analysis is one effective way to prove the absence of dead code.

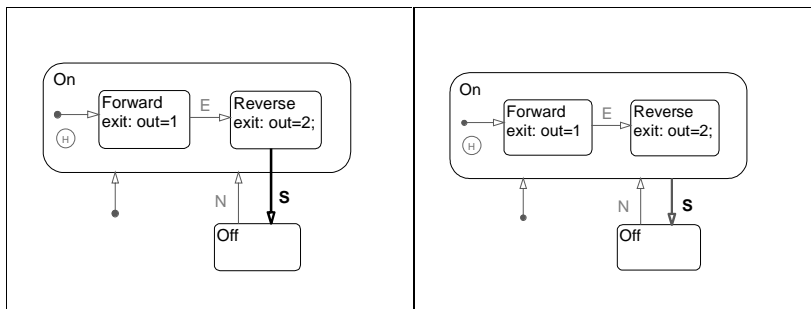


Figure 10: An example of state diagram logic dependent on context. On the left the transition labeled S explicitly exits the state Reverse. On the right the transition labeled S will implicitly cause either the Forward or Reverse state to exit requiring that logic is added to determine which exit action 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 is not available until the code is produced and can be very useful for identifying gaps in the specification where the designer and implementers may have different ideas of the correct behavior.

7.1 Measuring Requirements Completeness

Requirements that relate to input/output behavior can be expressed as a constraint on a system output for a particular input. When all of these test cases are executed they will generate a certain measure of model coverage.

To understand how requirements are measured for completeness using model coverage, consider a hypothetical design of a cruise control system. The inputs and outputs to the controller are shown in Figure 11. The controller uses sensor input for the brake pedal, accelerator pedal and vehicle speed. User input is generated from a Power switch, and Set, Resume, Increment, and Decrement buttons. The controller produces a throttle command used as a set point to the mechanical system that controls the throttle plate. The target speed for the controller also serves as an output for verification even though it is not required by the other system components.

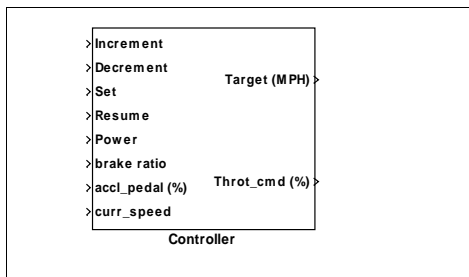


Figure 11: The cruise control input and outputs

The controller has the ability to adjust the target speed with an increment and decrement button. A list of functional requirements for the cruise control is shown in Table 1.

Table 1: A list of functional requirements for the cruise control

1	When the cruise control is powered on it shall enter an idle mode until a target speed is established that enables active control.
2	When the Set button is depressed while the cruise control is on it shall set the target speed to the current vehicle speed.
3	When the Resume button is depressed it shall set the target speed to the last value set by the vehicle speed since the control was powered on.
4	Pressing and releasing the Inc button in less than 1 second when the control is active shall cause the target speed to increase by 1 M.P.H
5	Holding the Inc button depressed when the control is active shall cause the target speed to increase by 1 M.P.H. every second.
6	Pressing and releasing the Dec button in less than 1 second when the control is active shall cause the target speed to decrease by 1 M.P.H
7	Holding the Dec button depressed when the control is active shall cause the target speed to decrease by 1 M.P.H. every second.
8	When the cruise control is not actively controlling speed, the throttle position shall be set to the same value as the accelerator pedal.
9	When the brake pedal is greater than zero and the cruise control is active the cruise control shall enter the override mode.
10	When the controller is in the override mode and the Set or Resume button is depressed the controller shall return to active control

An executable specification for the controller is implemented using Simulink and Stateflow. The top level diagram is shown in Figure 12. The only logic in this diagram is the switch and max blocks. The max block handles the case where the user commands a greater throttle than the cruise control by overriding the controller output with the commanded position. The switch block ensures that the accelerator pedal position is used as the commanded throttle position whenever the controller is not active.

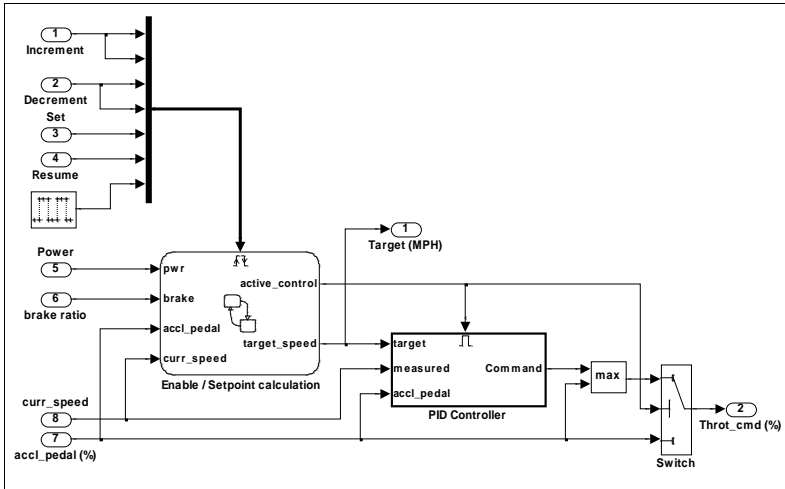


Figure 12: The Simulink block diagram of the cruise control.

The algorithm for adjusting the throttle command so that the vehicle speed matches the target speed is contained in the "PID Controller" block in Figure 12 and its contents are shown in the block diagram in Figure 2. The logic to enable and disable the block diagram and to determine the appropriate target speed is implemented in the Stateflow diagrams shown in Figure 13 which are contained in the "Enable / Setpoint calculation" block in Figure 12. On the left the logic is split into two parallel states, the top state determines the target speed by processing the events from the increment, decrement, set, and resume buttons and the bottom state determines when the controller is active based on the power setting, brake position, and button inputs. The contents of the hold state are drawn in a separate diagram to simplify the higher level diagram. The transition paths within the hold state describe how the Set and Resume events are processed.

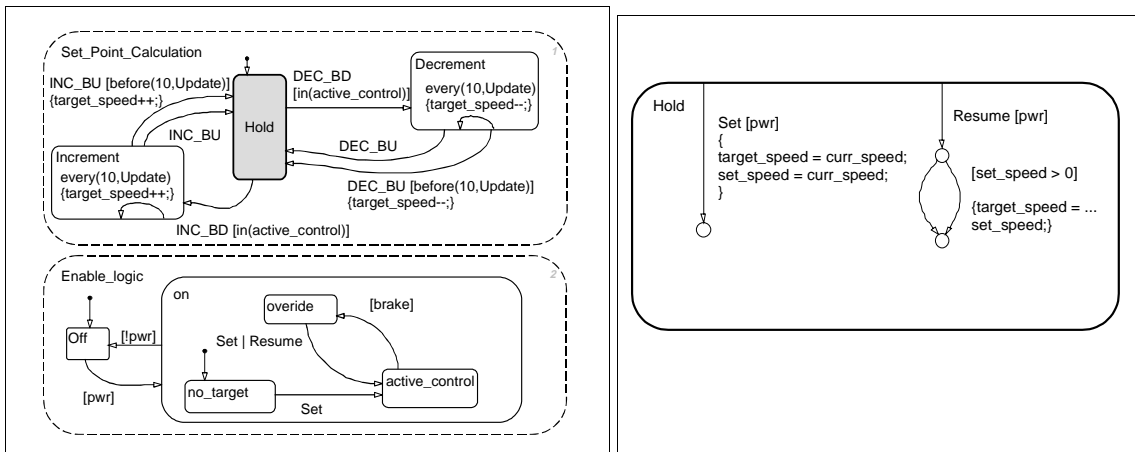


Figure 13: The state diagrams for the cruise controller. The top level diagram on the left is contained within the "Enable / Setpoint calculation" block in Figure 12. The diagram on the right shows the contents of the "Hold" state on the right.

Requirements are tested by deriving one or more scenarios where the required conditions are met and the output verified. As an example consider requirement 1. We test this requirement by first making a step change in the power signal from 0 to 1 and confirming that the control is not active. After a short delay we can simulate the Set button being pressed and confirm that the control becomes active. A plot of the input values over time is shown in Figure 14 with the expected value of target speed. This test case covers the first two requirements because the first requirement is dependent on a mechanism for setting a target speed.

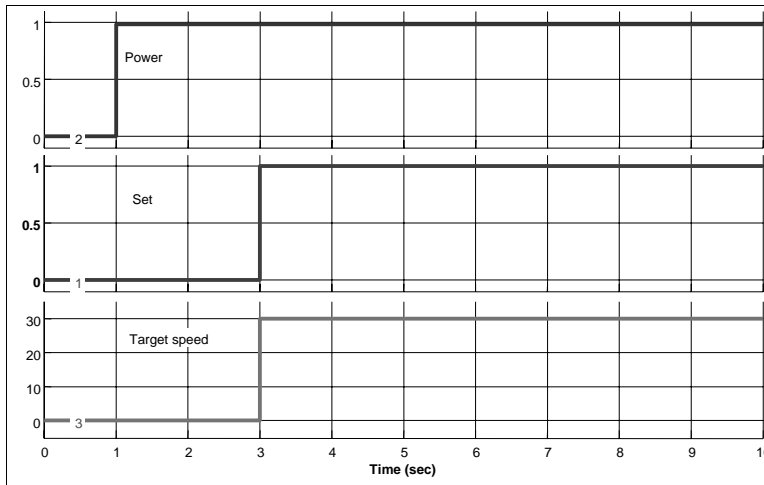


Figure 14: A plot of the time varying signals in the test case for requirements 1 and 2. The Increment, Decrement, Resume, Brake and Accelerator inputs are held at 0 and the speed input is held at 30 throughout the test.

A similar set of test cases can be developed for the remaining requirements. After the requirement-based test cases have been specified they are executed and a coverage report is generated. The portion of the coverage report relating to the transition "DEC_BD [in(active_control)]" is shown in Figure 15. The last line of the "Condition Coverage Details" indicates that the condition "in(active_control)" was always true when the transition was tested. To get complete coverage we would need a test case where the decrement button is depressed while the control is not active. Notice that this particular scenario is not described in any of the requirements in Table 1.

In this example the requirement-based test cases covered 90% of the model decisions and 88% of the model conditions. The uncovered items have been extracted from the report into Table 2 to conserve space. They are shown with a simple explanation of their design significance. In all of the missing cases the model may behave correctly but since the behavior is not specified in a requirement it represents a designers intuition or best guess. Realistically these cases may be the most important for testing the robustness of the system.

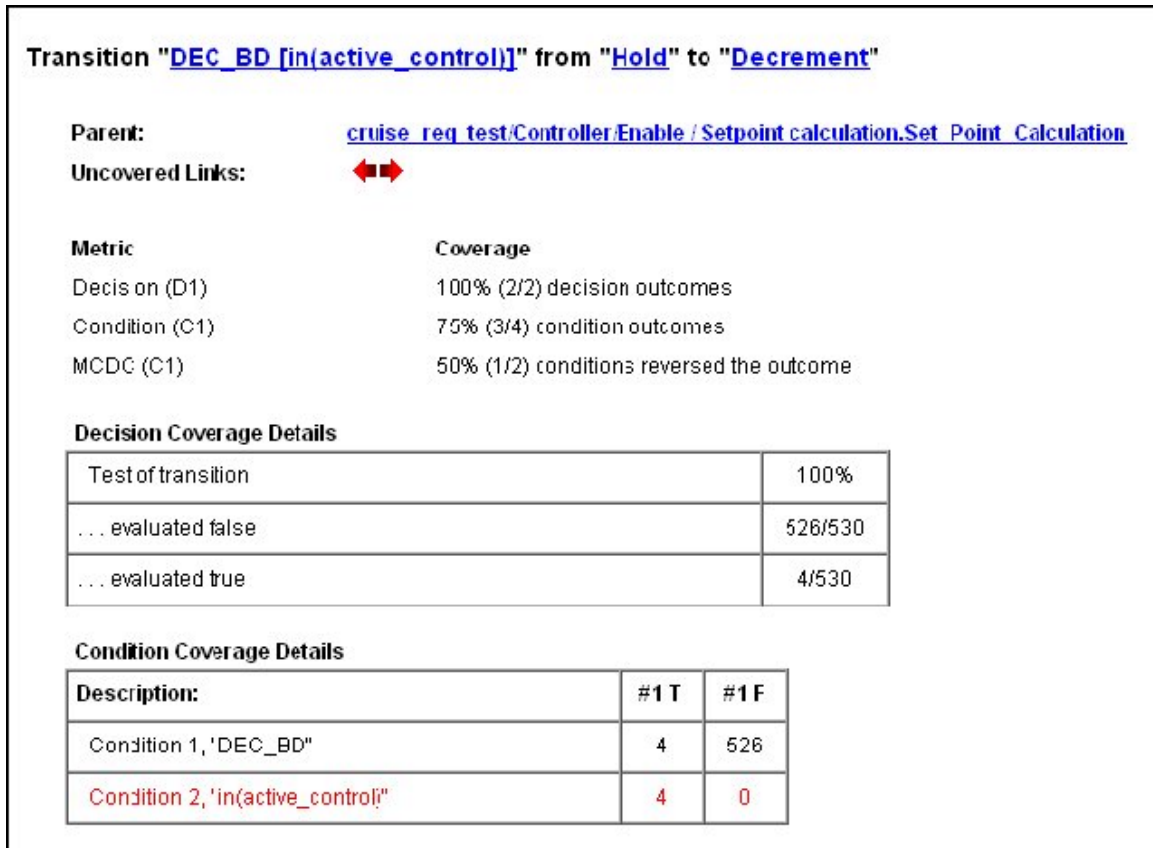


Figure 15: A portion of the coverage report after executing test cases for each requirement. The condition "in(active_control)" was never false. This means that the requirements never described what must happen when the decrement button is depressed while the cruise control is not in active control.

Table 2: Items not covered by any requirement-based test.

Object	Missing Coverage	Design Significance
Transition "[!pwr]"	was never true	Controller was never powered down
State "on"	never exited while in no_target	Controller was never powered down before a target speed was established.
	never exited while in active_control	Controller was never powered down while actively controlling.
	never exited while in override	Controller was never powered down from the override mode
Transition "DEC_BD [in(active_control)]"	in(active_control) was never false	Never depressed decrement while the control was inactive
Transition "INC_BD [in(active_control)]"	in(active_control) was never false	Never depressed increment while the control was inactive
Transition "Set [pwr]" ...	pwr was never false	Never pressed set while the controller was powered off
Transition "Resume [pwr]"	pwr was never false	Never pressed resume while the controller was powered off
Transition "[set_speed > 0]" ...	transition was never false	Never pressed resume when there was no valid speed target.

Much better coverage from the requirement-based test cases can be achieved with a precise understanding of what is implied. Adding the phrase "otherwise the button will have no effect" to the end of requirement 2 implies a test case in which the set button is depressed and the controller is not in the on state. In fact test cases can be generated that invoke the Set event from every state that is exclusive of the state "on".

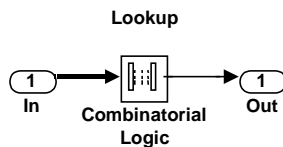
7.2 Coverage Analysis for Model Based Implementations

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.

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. Coverage analysis of hand written source code provides a useful view of test execution and can indicate problems in the code or limitations in the test suite. If the same analysis were presented on the list of machine instructions it would have almost no utility for identifying these problems because the semantics of the machine code are related to the processor instruction set and the design significance is not obvious.

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 16. 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.



Test of the logical index value	25%
... calculated to 0 based on inputs FF (output row 1)	0/101
... calculated to 1 based on inputs FT (output row 2)	101/101
... calculated to 2 based on inputs TF (output row 3)	0/101
... calculated to 3 based on inputs TT (output row 4)	0/101

```

/* CombinatorialLogic Block: <S2>/Combinatorial Logic */
{
  int_T rowidx=0;
  /* Compute the truth table row index corresponding to the input */
  rowidx = (rowidx << 1) + (int_T)(code_comp_P.Constant_Value[0] != 0.0);
  rowidx = (rowidx << 1) + (int_T)(code_comp_P.Constant_Value[1] != 0.0);
  /* Copy the appropriate row of the table into the block output vector */
  code_comp_B.Combinatorial_Logic =
    code_comp_P.Combinatorial_Logic_table[rowidx];
}

```

Figure 16: 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.

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

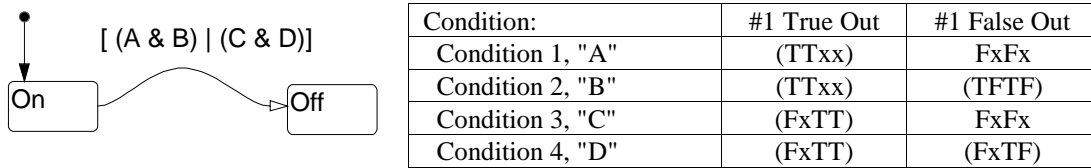


Figure 17: MC/DC coverage information for a transition with four conditions. The coverage information lists the specific combinations to satisfy MC/DC coverage.

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.

The goal of equivalency testing is not to show that the target is free of faults but simply that any fault in the target also exists in 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 18 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.

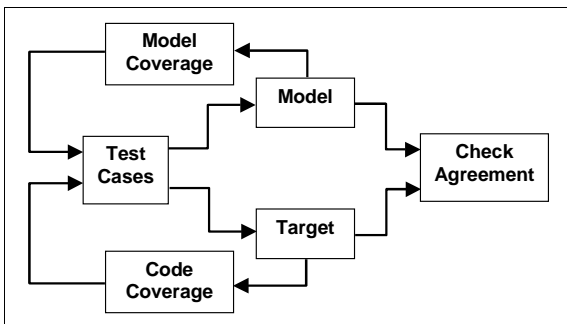


Figure 18: A diagram showing the iterative process of testing model and target equivalency. The top loop and bottom loop should be iterated until full coverage is achieved for the model and target. If the outputs agree in all test cases the goal has been achieved.

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. If the generated code has extraneous logic it is likely that it will not be tested to the same degree as the model, and the coverage will be less. If there are omissions in the generated code the coverage may be greater (this situation will more likely be detected by a test failure).

Unfortunately, inconsistent coverage results do not necessarily imply extraneous elements or omissions in the generated code. 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 19. 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.

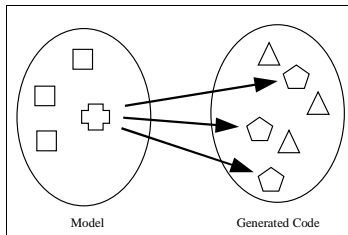


Figure 19: An Inline optimization where a single re-usable component in the model is replaced by a set of distinguishable constructs in the generated code, one for each use in the model.

The opposite of an inline optimization is a re-use optimization, shown in Figure 20. 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.

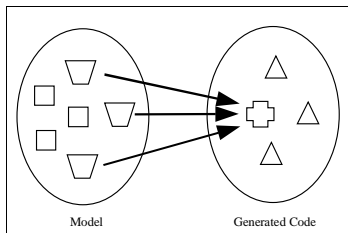


Figure 20: A Re-use optimization that takes a set of equivalent but distinguishable design constructs in a model and replaces them with a single re-usable construct in the generated code.

Another type of optimization that changes coverage is dead code removal, as shown in Figure 21. 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.

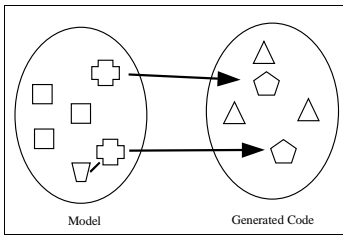


Figure 21: A dead code optimization where part of the model does not affect the generated code because that portion is non-functional.

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 22 in which the code on the left has three decisions while the code on the right has a single decision. While these differences can be minimized by using a combination of condition coverage and decision coverage there are similar permutations that cause condition coverage to disagree.

```

if ( C1 ) {
  if ( C2 ) {
    func1();
  } else {
    if ( C3 ) {
      func1();
    }
  }
}

if ( C1 && ( C2 || C3 ) ) {
  func1();
}

```

Figure 22: An example of two functionally identical code fragments having different coverage requirements based on the coded structure.

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.

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 which 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] Beizer, B., "Software Testing Techniques," (Van Nostrand Reinhold, New York, 1990) Second Edition.
- [3] Chilenski, J. J. and Miller, S. P., "Applicability of modified condition/decision coverage to software testing," Software Engineering Journal, September, 1994.
- [4] Poston, R. M., "Automating Specification-Based Software Testing," (IEEE Computer Society Press, Los Alamitos, 1996)
- [5] Stateflow User's Guide, Version 4.0, The MathWorks Inc., September, 2000.
- [6] Using Simulink, Version 4.0, The MathWorks Inc., November, 2000.