2013-01-2237

Requirements Modeling and Automated Requirements-Based Test Generation

John Lee and Jon Friedman
MathWorks

ABSTRACT

For many mission critical systems, demonstrating that all requirements have been met via a set of requirements-based tests is often mandated by internal processes or external standards. Traditional coverage approaches, however, do not address this mandate because they measure only the coverage of the design by determining which paths in the design have been executed. Determining whether a given set of test vectors covers the design requirements (as opposed to merely covering the design) is a challenge.

Creating a set of test vectors to cover the requirements can be difficult and time consuming. Using techniques first identified in the 1970s and modern Model-Based Design tools, we present a novel approach, based on work presented in [2], to automatically generate a set of requirements-based test vectors. In this paper, we discuss how requirements captured in a natural language can be modeled using Cause-Effect graphs, introduced in [1]. We then translate the Cause-Effect graph into Simulink® and Stateflow® to identify conflicting requirements and automatically generate a set of test vectors that can be assessed for completeness using coverage objectives such as Modified Condition/Decision Coverage (MCDC). We apply the same test vectors to the design model, which can be developed independently from the original requirements. This approach enables engineers to determine if their design is sufficiently covered by the set of requirements-based test vectors. Any parts of the design not covered must be investigated further to determine if the design elements should exist or if there is a problem with the requirements.

INTRODUCTION

The software that runs mission critical systems on modern automobiles, aircraft, and spacecraft is composed of millions of lines of code. Verifying that such software meets its design requirements is a significant challenge. To test software, engineers typically employ requirements-based and/or coverage-based methodologies.

In requirements-based testing, a set of test vectors and expected outputs are developed based on the requirements. These tests typically are written with the goal of fully testing all of the requirements. However, the completeness of the tests is difficult to measure because some requirements can be verified by a single test while others require a set of tests to verify. Moreover, even when a full set of requirements-based tests are applied to a design, this does not guarantee that the entire design has been tested or that there is no unintended functionality in the design.

In coverage-based testing, testers develop a set of test vectors with the goal of fully executing the design so that all design functionality is excited and unreachable portions of the design are identified. However, creating a set of test vectors that fully execute the design can be time consuming and does not guarantee that all of the requirements have been met. It can also be challenging to trace test vectors to the original requirements to identify which requirements have been tested. One way to have the best of both approaches is to create a set of test vectors that cover the requirements and measure whether these tests execute or cover the design based on a criterion of coverage such as MCDC.

For a number of years, engineers have sought to use models to capture requirements in a style that is unique and distinct from the design models traditionally used in Model-Based Design. Such models are often referred to as *requirements models* to differentiate them from *design models.*

The term *requirements model* has several definitions, including some closely related to the system level architecture. For the purpose of this paper, *requirements modeling* refers to how to model the input-output relationship expressed in requirements via "shalls". A simple example of a "shall" is: *When A occurs, then the system shall B*. Requirements models typically fall into one of two broad groups:

- Input-output descriptions, in which only the input and output are described in the model
- Higher level abstraction descriptions, in which a high level or abstract dynamic or logic model of the

system is included in the requirements model and the expected output can be computed from the requirements model, based on the input.

The goal of requirements models is to capture the functional requirement in a clear, concise, analyzable and executable manner, which is typically not possible with natural language. The requirements models can then be used to evaluate the interaction and compatibility of requirements from disparate sources as well as to develop tests and acceptance criteria (or expected outputs). The use of the requirements models for test creation enables engineers to assess the completeness of the tests using different notions of coverage on the requirements model such as Function coverage, Statement coverage, Decision coverage, Condition coverage, or Modified Condition/Decision Coverage (MCDC).

Requirements models have traditionally been less detailed models of the desired behavior that can be elaborated and developed into design models by design engineers [4,5]. This approach relies on the interpretation of the engineer to convert the natural language requirement into the abstract level design, which can result in a single point failure if the interpretation is incorrect. Recently, engineers have begun to develop models of the requirements that capture only the relationship between the test and the expected output (or input-output relationship). More importantly these models are developed independently from the design models *by different engineers using different modeling styles*. The potential single-point interpretation failure mode is reduced because one group of engineers develops requirements models from which the tests and expected outputs are generated and another group develops design models to which these tests can be applied and the independent results can be compared as shown in Figure 1.

Several modeling styles are available for capturing the input-output relationship of a requirement. This paper focuses on how requirements expressed in a natural language can be modeled using Cause-Effect graphs coupled with the expected output, to create a requirements model. The relationship between the test input and expected output can be captured by an additional portion of the requirements model that may contain a simple input-output relationship or abstract models of the dynamics or logic, so that the input-output relationship can be computed. The portion of the requirements model which captures the expected outputs is called the *test oracle*.

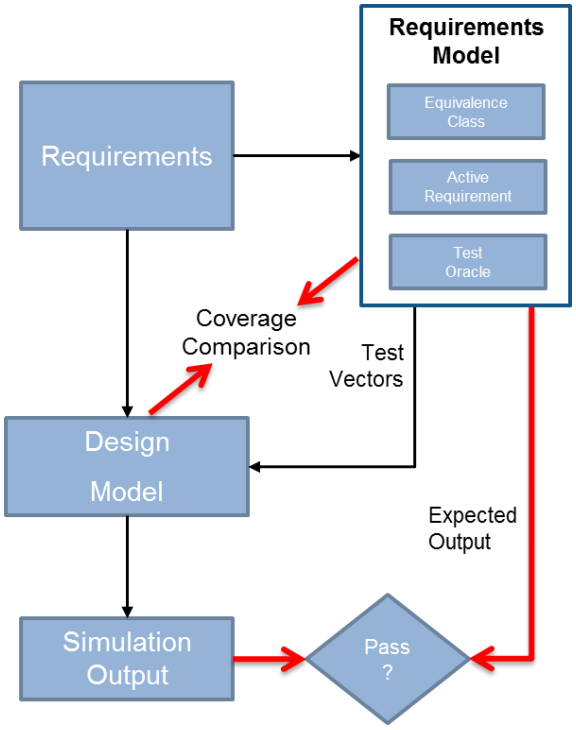


Figure 1: Workflow comparing results of design and requirements models.

In the paper, we also describe how the Cause-Effect graphs can be captured in models using Simulink and Stateflow and then analyzed using Simulink Design Verifier™ to create a set of test vectors to achieve requirements model coverage objectives, such as Condition, Decision, and Modified Condition/Decision (MCDC).

CAUSE-EFFECT GRAPHS

Cause-Effect graphing is a modeling method, introduced in 1979 by Glenford J. Myers in The Art of Software Testing, through which a natural language specification can be formally mapped. Cause-Effect graphing maps inputs to outputs with a set of constraints. In this context, a *cause* is a distinct input condition or equivalence class of input conditions, and an *effect* is an active condition or applicable requirement. In a Cause-Effect graph, inputs (causes) and requirements (effects) are the nodes of the graph. Typically, the graphs are drawn with causes on the left, effects on the right, and the logical relationships between them graphed on lines using the notation shown in Figure 2.

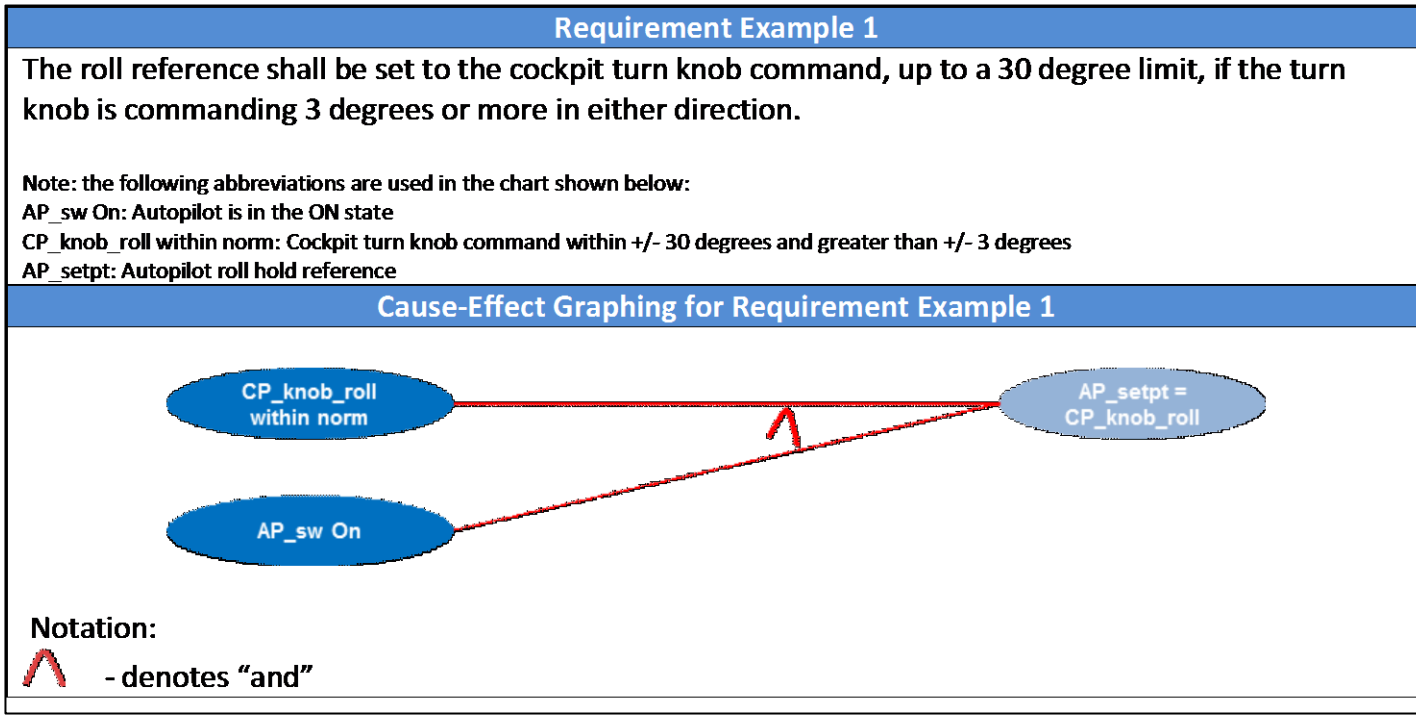


Figure 2: Cause-Effect Graph of a Requirement

In Figure 2, the Cause-Effect graph provides a diagrammatic view of the relationship between input (cause) and expected output (effect) specified by the requirement. The Cause-Effect graph can be used to derive test cases. For the above example, test cases are created based on one input condition (CP_knob_roll within norm) and a second input condition (AP_sw On) resulting in an effect (AP_setpt = CP_knob_roll). The advantage of Cause-Effect graphing is that the graphs enable the tester to formally represent the requirements and systematically create test cases based on those requirements.

For this paper, we applied Cause-Effect graphing to develop a set of requirements based tests using the requirements introduced in [4]. This example has a design model that the author developed based on the requirements. We will apply test cases developed using Cause-Effect graphs to the design model from the original paper to assess the completeness of the tests on the design. Our goal is to use techniques as outlined by [2] to gain greater insight on the benefits of using requirements modeling in a Model-Based Design environment.

EXAMPLE – AUTOPILOT ROLL CONTROL

In [3], the author introduced a set of natural language functional requirements for an aircraft autopilot. In this example, these requirements are used to illustrate how Cause-Effect graphs can be applied to manually create a requirements model and how this model can be used to automatically generate requirements-based test vectors.

When considering a single requirement, this approach seems simple to implement. However, the systems that engineers design today have thousands of requirements, making the task significantly more complex. As shown in Figure 3, the Cause-Effect graphing for the requirements document in [3] is much more complicated than the example shown in Figure 2. As the number of requirements increases along with their complexity and the intricacy of the interaction between requirements, it is easy to see the challenge of creating sufficient and meaningful test cases to verify that a given design meets all requirements—especially since each transition line can represent multiple test cases.

Structured methodologies such as Cause-Effect graphing provide a basis for systematically developing and tracking test cases, but the process of manually creating test cases can be tedious. State charts can be used to address this issue. For example, the manually created Cause-Effect graph shown in Figure 3 can be represented by an equivalent flowchart or stateless state chart using Stateflow. The Cause-Effect representation shown in this paper adopts the modeling style developed by John Deere [2]. This particular flowchart modeling style was developed to work with Simulink Design Verifier for test case generation.

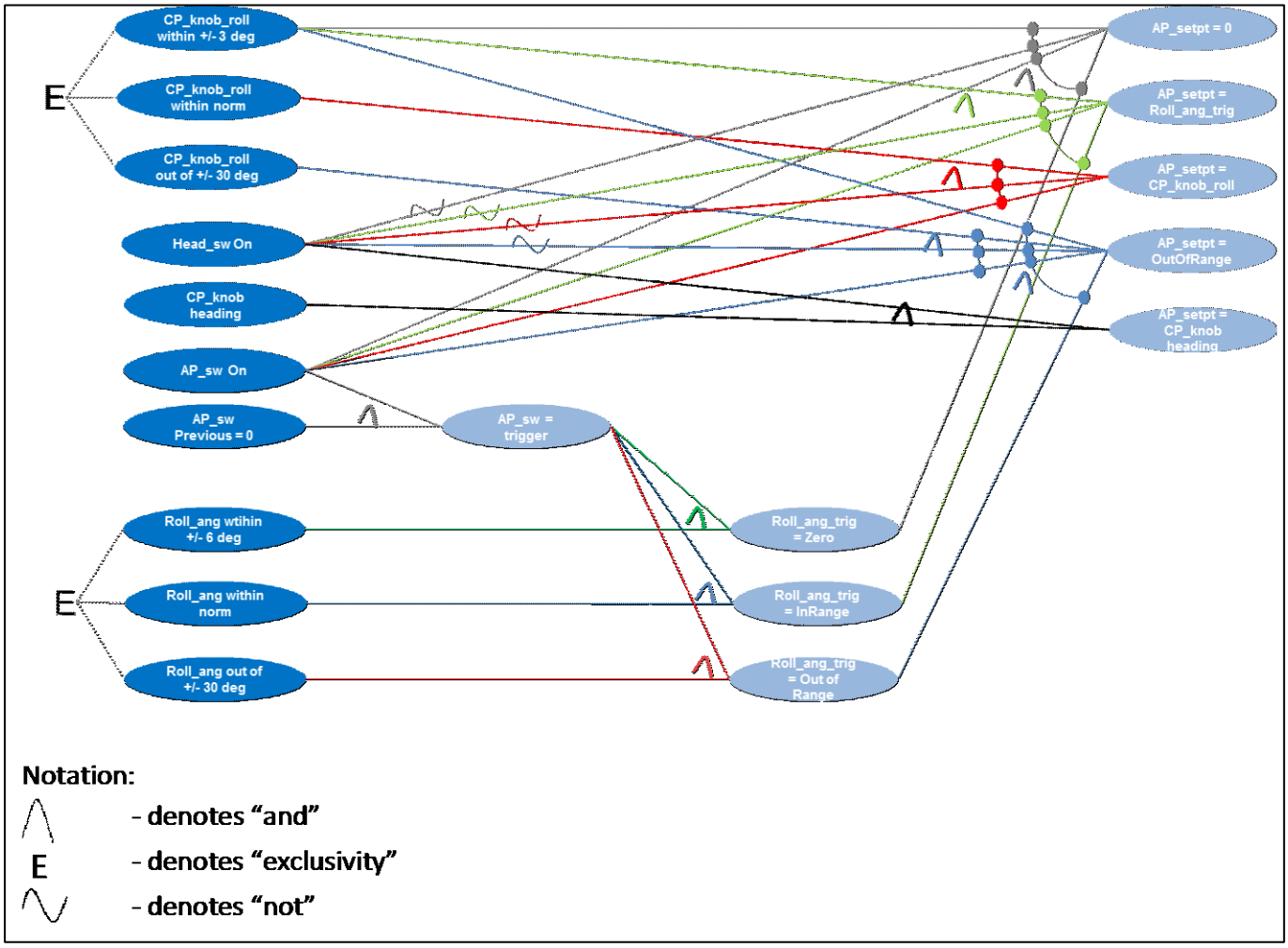


Figure 3: Cause-Effect Graph of Auto Pilot Example

Here are some key characteristics of this modeling style (extending the example Cause-Effect graph shown in Figure 2):

1. The Cause-Effect graph is split between two different models, shown in Figure 4. We call the "cause" and "effect" portion of the model the *reference model*, and the "expected output" portion the *test oracle*.

2. The reference model is further divided into two sections, shown in Figure 5. The first section parses the input signals into equivalence classes, and the second section takes the equivalence class inputs and determines which of the requirements are being activated. In this example, one equivalence class is the Cockpit turn knob command within +/- 30 degree and greater than +/- 3 degrees. Then the requirement being activated is that "the roll reference equal Cockpit turn knob value," which is TRUE when the equivalence class defined above and AP (Autopilot) switch ON are both TRUE.
3. Based on the active requirement, the test oracle then calculates the expected output. In this example, the expected output of the Autopilot set point is equal to the Cockpit turn knob command when the corresponding requirement is TRUE, as shown in Figure 6.
4. Both the reference model and the test oracle use flowchart syntax to model the cause and effect relationship.

Once the requirement model is completed, it is augmented with blocks from the Simulink Design Verifier to better constrain the test vectors.

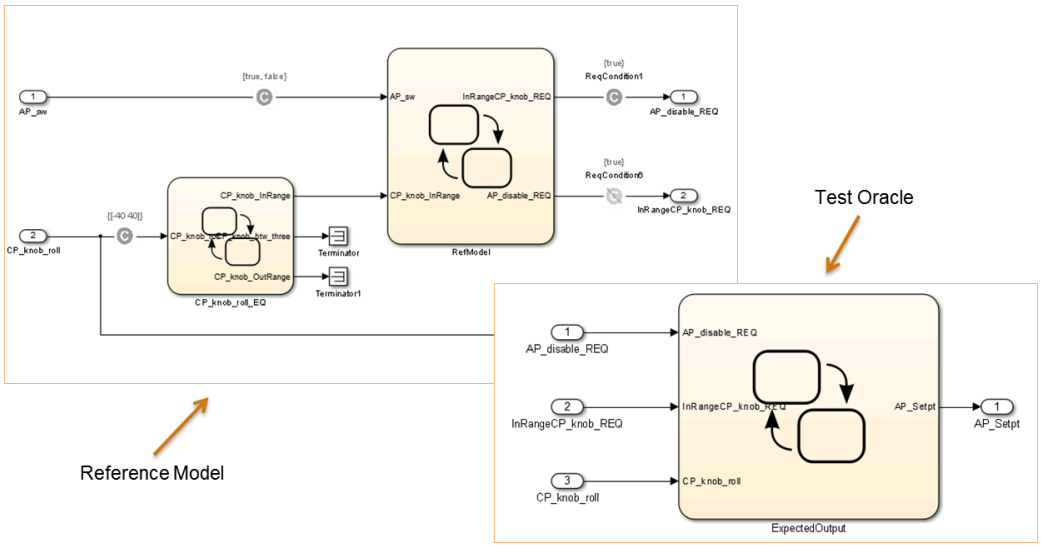


Figure 4: Structure of Cause-Effect graph

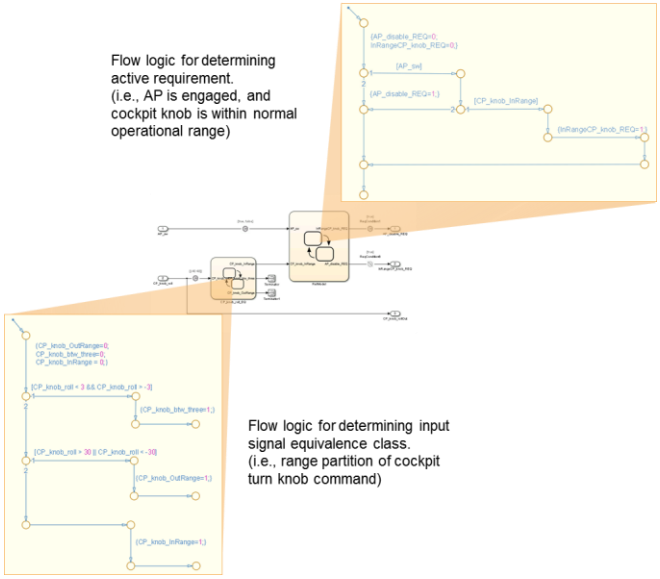


Figure 5: Equivalence Classes and Active Requirements

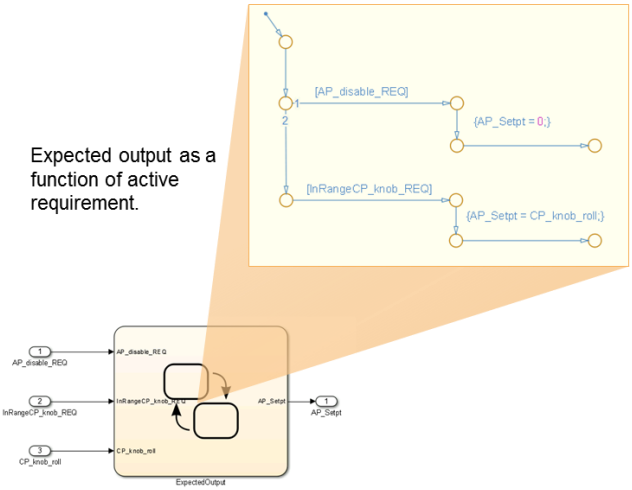


Figure 6: Expected output calculated by the test oracle

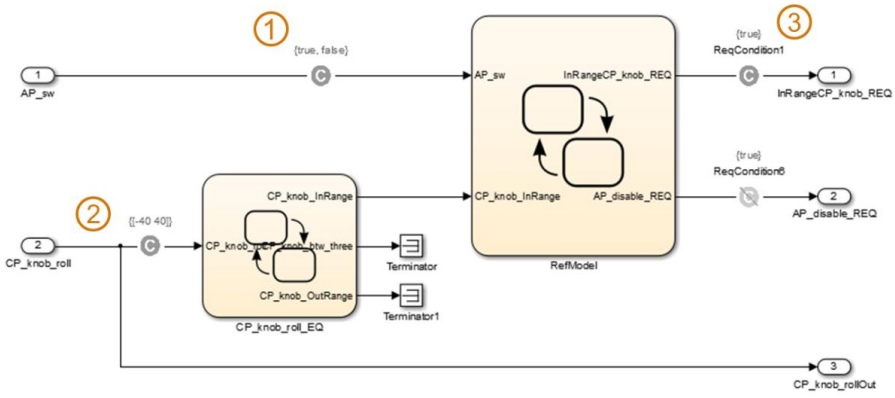


Figure 7: Additional Constraint or "C" Blocks

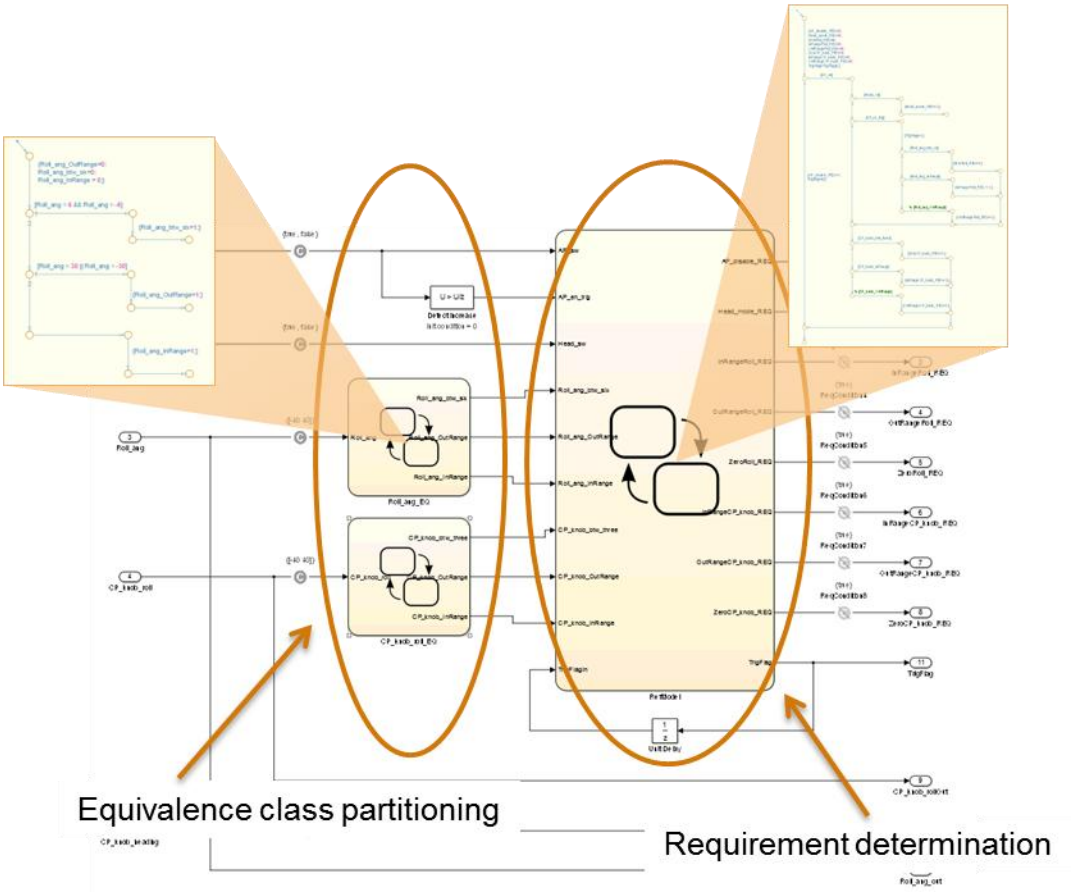


Figure 8: Requirements Model

In Figure 7 the "C" blocks on the signal lines are used to achieve the following:

1. The "C" block labeled "1" controls the input signal such that it can only be either *True* or *False*.
2. The "C" block labeled "2" controls the input signal such that it can only be in the range of -40 to +40 degrees.
3. The "C" block labeled "3" enforces that the requirement associated with that signal (AP_disable_REQ) will always be *True* during test generation. Because this output is always set to *True*, conflicting requirements will show up as an "unreachable" path.

Continuing the autopilot example shown above from [3], the Cause-Effect graph of the autopilot requirements is manually translated into a model using Stateflow and used as input to generate test cases using Simulink Design Verifier.

The flowchart reference model of the autopilot requirements graphed in Figure 3 is shown in Figure 8 and the test oracle is shown in Figure 9.

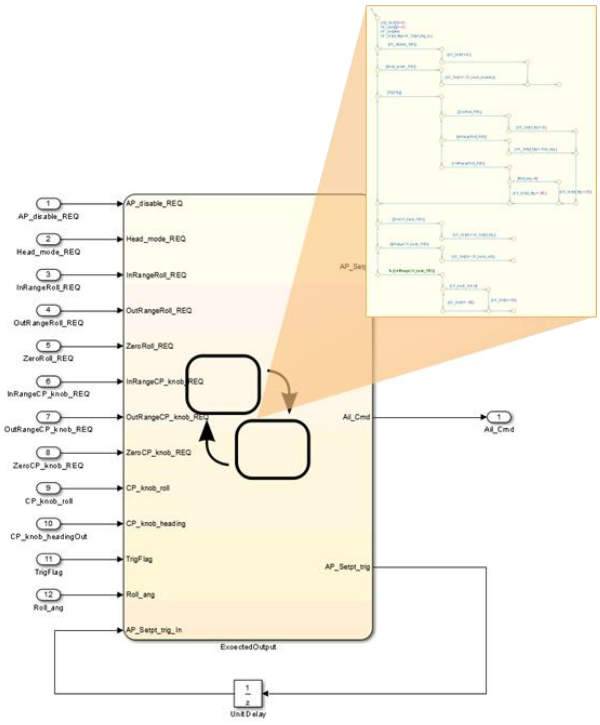


Figure 9: Test Oracle

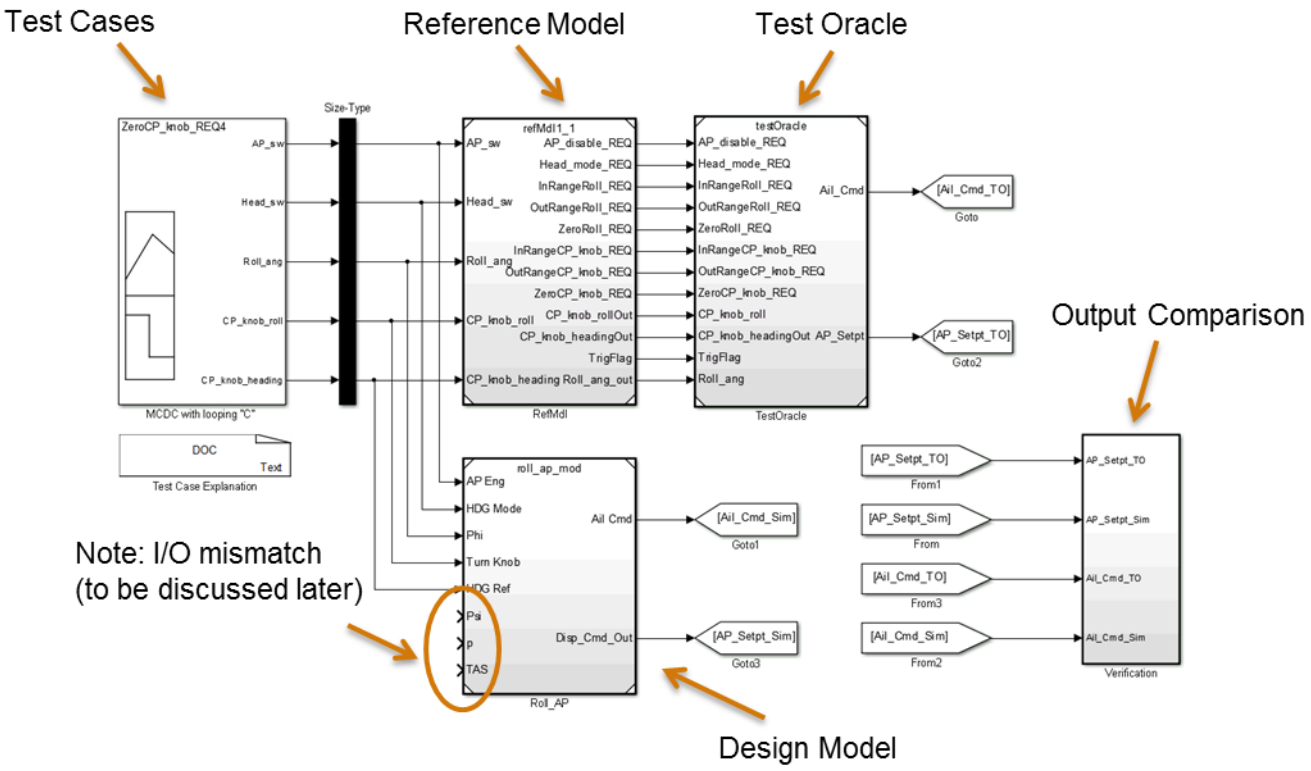


Figure 10: Test Harness Model

Once complete, the reference model can be used as input to Simulink Design Verifier to generate test cases. The type of test cases can be customized via special property blocks, which add constraints to the model. The "C" block described above, and shown in Figure 7, is an example of such a constraint. For example, a standard set of test cases can be generated to achieve model coverage objectives of the requirements model. Additionally, a customized constraint can be added to the model to provide a specification on a signal's initial condition, and the test vectors can be generated to satisfy this constraint.

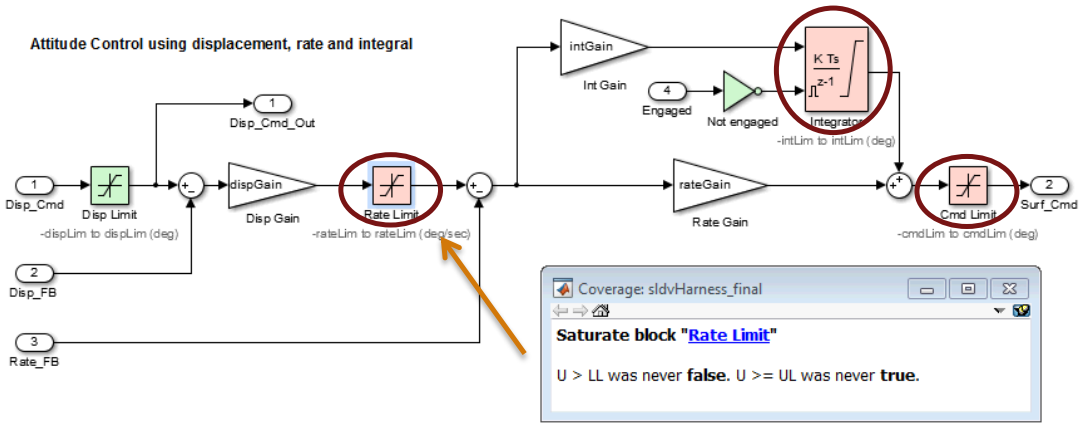
A more rigorous approach, as suggested in [2], is to create a set of MCDC test cases, while holding each of the reference model outputs (individual requirements) *True*. This approach provides two important advantages:

- It draws attention to potential errors. If a requirement (or objective) on a branch of the requirements model is unreachable, then this requirement is mutually exclusive from the others on the branch. A review of such a result would highlight potential error that may occur while creating the model.
- It systematically creates more test cases, which can identify errors that might otherwise have been missed using a manual test creation process. Since the test oracle provides the expected output or results, simulation results from these test cases can be automatically or programmatically analyzed allowing engineers to focus on only those test cases where there is a discrepancy between the two models).). As a result, running additional test cases, including field recorded data and random noise tests, will not add significant analysis time.

Once test cases have been created, they are placed inside a test harness model along with the reference model and test oracle, as shown in Figure 10. The same test vectors fed into the requirements model are then fed into the design model and the outputs are compared.

The output of the test oracle and design model should be identical. If they are not, then there is a discrepancy between the requirement and the design. A discrepancy does not imply that the requirement is not met, but each discrepancy does need to be investigated. The requirements model might itself contain errors, just as the design model can, since they are both built manually. The key point is that the probability of both independent modelers making the same error with different modeling approaches at the same point is low.

As a final check, engineers can run the generated test vectors on the design models and measure the model-level coverage. Since the test cases were generated from the requirements model, it can be reasoned that if the requirements model was 100% covered by the test vectors then applying them to the design model should result in 100% coverage of the design model. If it does not, then some unintended functionality may



Note: blocks circled have missing coverage information

Figure 11: Feedback Algorithm

have been included in the design. Failing to achieve 100% coverage may also point to incomplete requirements or missing derived requirements.

RESULT

By comparing the coverage analysis between the requirement model and design model in [3], we found the following issues:

1. Missing coverage objective.
   Applying the set of automatically generated test cases from the requirements model to the design model revealed missing coverage in the design model. Upon further analysis it was determined that three high level requirements, listed below, were not detailed enough to create the requirements model with sufficient content to match that of the design model.
   - Steady state roll commands shall be tracked within 1 degree in calm air.
   - Response to roll step commands shall not exceed 10% overshoot in calm air.
   - Small signal (<3 degree) roll bandwidth shall be at least 0.5 rad/sec.

   To meet the performance requirements listed above, the design model must include a closed-loop feedback algorithm, as shown in Figure 11. The need for the added structure was identified when the test cases were generated from the requirements model and several blocks were not covered (see Figure 11) and three inputs were missing (see Figure 10). Since the existence of the closed-loop feedback is not specified in the system-level requirements, both the feedback signals (inputs) and the elements of the feedback algorithm were not covered by the tests generated from the system level requirements model. The uncovered elements include logic and saturation blocks circled in Figure 11. When the additional requirements for the structure of the feedback are added, the missing inputs are generated using the approach discussed in this paper.

2. Test oracle and design model output mismatch.
   The output of the test oracle and the design model were different under the following conditions:
   - Autopilot switch was engaged from time = 0.
   - Set point for autopilot while autopilot is not engaged.

   In this case, there is either an implicit assumption about the behavior at time=0 or some presumption of manual override. It is important to note that this approach highlighted the disconnect between the requirements and the design for these conditions.

CONCLUSIONS

We draw the following conclusions from this work:

- The modeling style used for Cause-Effect graphing is sufficiently different from those used in design models that it may provide independence between development and verification paths. For engineers, the use of this modeling style keeps the focus on requirements, and not on design or implementation.
- When generating test cases and holding each of the output signals to be *True*, various mutually exclusive requirements were deemed unreachable. This provides a systematic way of analyzing each requirement and its exclusivity with respect to other requirements. For example, the requirement states that the roll mode of Autopilot can only be active when no other lateral mode is active. This is verified when generating test cases

while constraining the roll mode to be *True*; in this case the heading mode switch logic path was shown as unreachable.

- Cause-Effect graphing can be one way to systematically track each requirement (and its interaction with other requirements) as one builds a graphical representation or a model of the requirement.
- Cause-Effect graphing can be time consuming. However, this initial time investment can be recouped via:
  - Avoiding potential rework at later development stages. Upfront requirements analysis can save thousands of hours on a typical industrial project [6].
  - Automatically generating tests from the requirements models.
  - Automating the review of test results via comparison with the test oracle.
- The Cause-Effect graphing method (and test generation), while applicable to system level requirements, may lead to less than 100% coverage of a design model due to derived requirements. As a result, the application of Cause-Effect graphing is sometimes more straightforward for low level requirements than for system level requirements. For high level requirements, the output usually has a greater tolerance, with many of the lower level requirements left to the interpretation of the algorithm designer. This in turn makes the coverage comparison more nuanced and output comparison more prone to acceptable differences. As a result, some engineers may focus their use of Cause-Effect graphing only on low level requirements. The requirement model may need to be elaborated—like the design model—throughout the development cycle.
- Cause-Effect graphing is one form of requirements modeling that can be paired in a straightforward manner with automatic test generation. An extension of the requirements model can be also be viewed from the perspective of test generation. Modeling constructs such as boundary values, signal initialization, and so on can be added, making it possible to generate test cases from the models with tools such as Simulink Design Verifier. This potential makes reuse of test vectors much easier and review and modification process much easier as well.
- Lastly, using the test oracle, the review of the test results with the expected outputs can be automated.

REFERENCES

1. Myers, Glenford J., The Art of Software Testing, 2nd Eddition, John Wiley & Sons, Inc., Hoboken, New Jersey, 2004
2. Ross, Jim, 2013 SAE World Congress Technical Track - AE316, “Requirement-based Test Case Generation and Coverage Analysis”
3. Potter, Bill, Model-Based Design for DO-178B 2008 MathWorks News Letter, http://www.mathworks.com/company/newsletters/articles/model-based-design-for-do-178b.html
4. Barnard, Paul, Graphical Techniques for Aircraft Dynamic Model Development, 2004 AIAA
5. Yang, J., Bauman, J., and Beydoun, A., "Requirement Analysis and Development using MATLAB Models," *SAE Int. J. Passeng. Cars - Electron. Electr. Syst.* 2(1):430-437, 2009, doi:10.4271/2009-01-1548.
6. Lin, Joy, Measuring Return on Investment of Model-Based Design, MathWorks White Paper, http://www.mathworks.com/model-based-design/mbd-roi-video/Measuring_ROI_of_MBD.pdf

CONTACT INFORMATION

All contact should be directed to Jon Friedman – email: jon.friedman@mathworks.com.