

Requirements Modeling and Automated Requirements-Based Test Generation

Chien-Chang Lee and Jon Friedman
MathWorks Inc.

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.

CITATION: Lee, C. and Friedman, J., "Requirements Modeling and Automated Requirements-Based Test Generation," *SAE Int. J. Aerosp.* 6(2):2013, doi:10.4271/2013-01-2237.

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*.

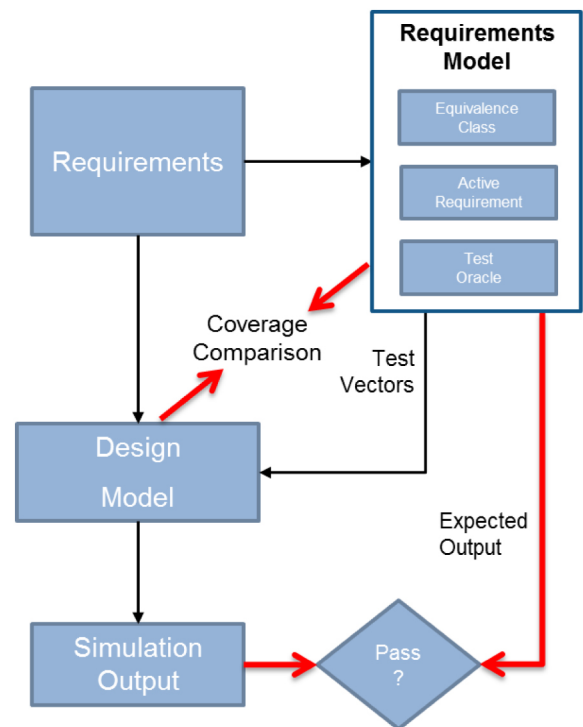


Figure 1. Workflow comparing results of design and requirements models.

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](#).

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 =

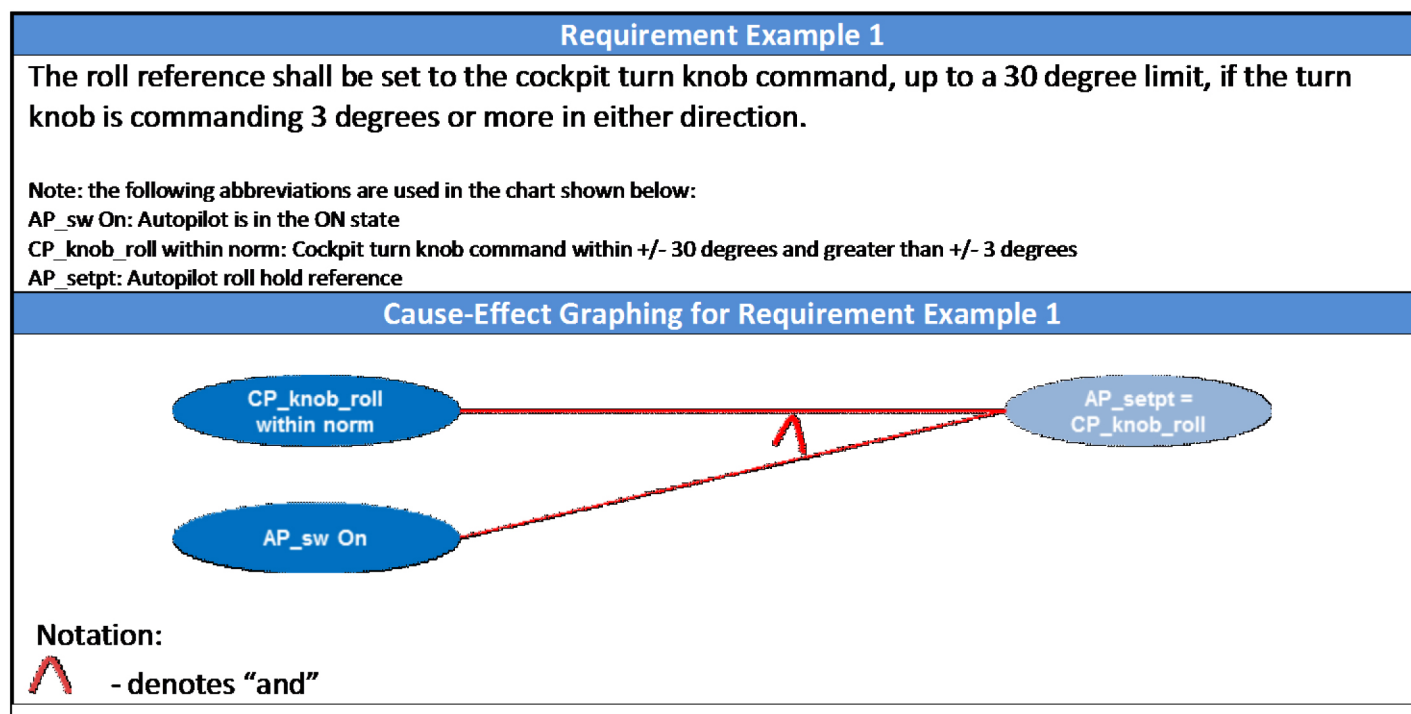


Figure 2. Cause-Effect Graph of a Requirement

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.

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