NASA/CR-2006-214307

Testing Strategies for Model-Based Development

Mats P. E. Heimdahl
University of Minnesota, Minneapolis, Minnesota

Mike Whalen
Rockwell Collins, Inc., Cedar Rapids, Iowa

Ajitha Rajan
University of Minnesota, Minneapolis, Minnesota

Steven P. Miller
Rockwell Collins, Inc., Cedar Rapids, Iowa

April 2006

NASA/CR-2006-214307

Testing Strategies for Model-Based Development

Mats P. E. Heimdahl
University of Minnesota, Minneapolis, Minnesota

Mike Whalen
Rockwell Collins, Inc., Cedar Rapids, Iowa

Ajitha Rajan
University of Minnesota, Minneapolis, Minnesota

Steven P. Miller
Rockwell Collins, Inc., Cedar Rapids, Iowa

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Cooperative Agreement NCC-1-01001

April 2006

Available from:

NASA Center for AeroSpace Information (CASI)
7121 Standard Drive
Hanover, MD 21076-1320
(301) 621-0390

National Technical Information Service (NTIS)
5285 Port Royal Road
Springfield, VA 22161-2171
(703) 605-6000

Testing Strategies for Model-Based Development

Mats P.E. Heimdahl
Mike Whalen
Ajitha Rajan
Steven P. Miller

heimdahl@cs.umn.edu
ajitha@cs.umn.edu
mwwhalen@rockwellcollins.com
spmiller@rockwellcollins.com

Department of Computer Science and Engineering
University of Minnesota
4-192 EE/SC Building
200 Union Street S.E.
Minneapolis, Minnesota 55455

Advanced Technology Center
Rockwell Collins, Inc.,
Cedar Rapids, IA 52498 USA

Abstract

Model-based software development offers new opportunities and challenges for validation and verification of safety-critical software. Since models have well-defined syntax and semantics, it is possible to test models as well as source code and to define structural coverage metrics over models. Further, given a formal description of requirements, it is possible to use automated tools to check whether models satisfy requirements and to describe objective notions of requirements coverage. Recently, model-based testing tools have emerged that allow auto-generation of test-cases given a model and structural coverage metrics. Nevertheless, there is a great deal of uncertainty as to which tools and techniques are effective and how to structure a testing process using model-based development.

In this report, we describe some of the issues in model-based testing, present an approach for testing artifacts generated by a model-based development process, and describe the relationship between this process and existing standards such as DO-178B. This approach divides the traditional testing process into two parts: requirements-based testing (validation testing) *which determines whether the model implements the high-level requirements and* model-based testing (conformance testing) *which determines whether the code generated from a model is behaviorally equivalent to the model. The goals of the two processes differ significantly and this report explores suitable testing metrics and automation strategies for each. To support requirements-based testing, we define novel objective requirements coverage metrics similar to existing specification and code coverage metrics. For model-based testing, we briefly describe automation strategies and examine the fault-finding capability of different structural coverage metrics using tests automatically generated from the model.*

Table of Contents

1 Introduction

Traditionally, software development has been largely a manual endeavor. Validation that we are building the right system has been achieved through requirements and specification inspections and reviews. Verification that the system is developed to satisfy its specification is archived through inspections of design artifacts and extensive testing of the implementations (Figure 1). In critical avionics applications, the validation and verification phase (V&V) is particularly costly and consumes a disproportionably large share of the development resources. Thus, if we could devise techniques to help us reduce the cost of V&V, dramatic cost savings could be achieved. The current trend towards *model-based* (or *specification centered* [8], [55]) development is one attempt to address this problem.

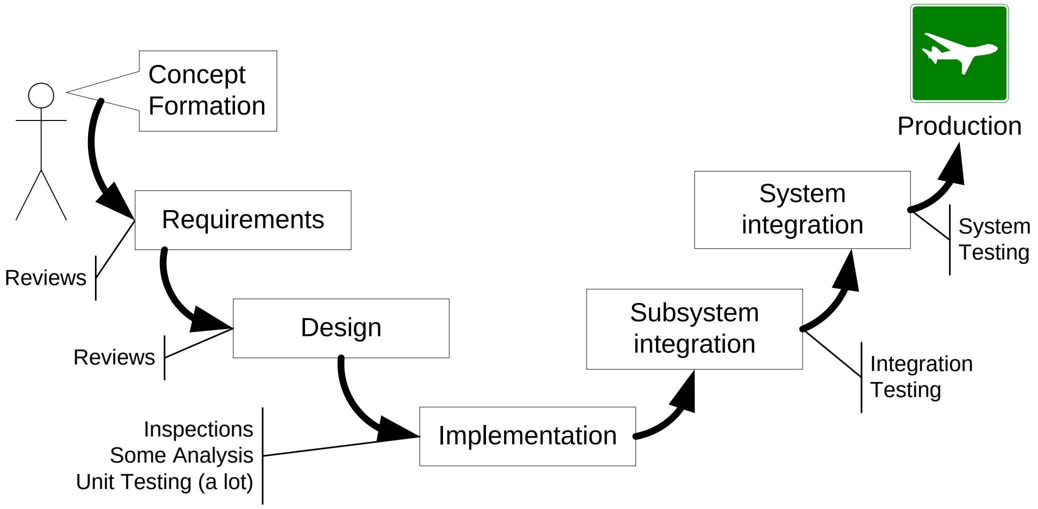

Figure 1: Traditional and largely manual software development.

1.1 Model-Based Development

In model-based development, the development effort is centered on a formal description of the proposed software system. For validation and verification purposes, this *formal specification* can then be subjected to various types of analysis, for example, completeness and consistency analysis [29],[32], model checking [23],[11],[13],[33], theorem proving [4], [7], and test-case generation [10],[21],[16],[9],[34],[44]. Ideally, through manual inspections, formal verification, and simulation and testing we convince ourselves (and the regulatory agencies) that the software specification possesses desired properties. The implementation is then *automatically generated* from this specification (Figure 2). There are currently several commercial and research tools that provide these capabilities. Commercial tools are, for example, SCADE Suite from Esterel Technologies [17] and Statemate from i-Logix [26]. Examples of research tool are SCR [31], RSML^{-e} [54], and Ptolemy [37].

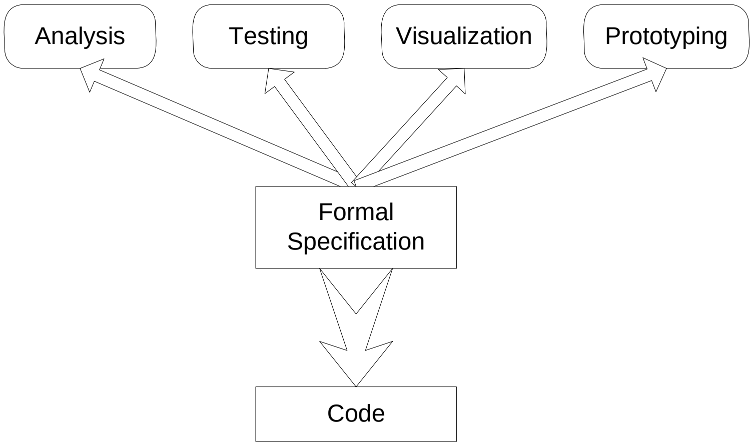

Figure 2: Model-Based Development.

The capabilities of model-based development enable us to follow a different development process. The development is now centered on the formal specification and the V&V has been largely moved from testing the code (Figure 1) to analyzing and testing the specification (Figure 3)—the traditional (and, in the critical systems domain, very costly) unit testing of code is largely replaced with testing and analysis of the specification in a hope to provide higher quality at lower cost.

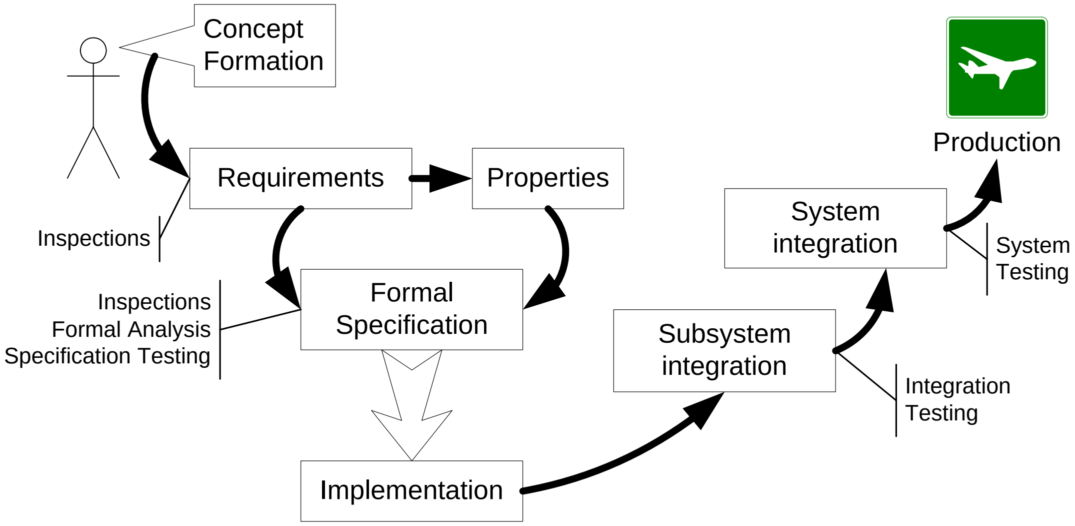

Figure 3: Model-based development process.

The focus on the specification and new approach to software validation and verification in model-based development raises several challenges. This report describes the opportunities and challenges related to testing in this new model-based world.

1.2 Testing and Test Automation

The new process enabled through model-based development divides the testing of a system under development into two distinct tasks. First, the formal model must be *validated* to ensure that it captures the behavior we actually want from our system, i.e., we must test the model to convince ourselves that it satisfies the “true” requirements. Second, we want to ensure that the code developed (or generated) from the formal model correctly implements the behavior described in the model, i.e., we want to show that the implementation *conforms* to its specification.

Figure 4 shows an overview of the testing needed to validate that the model is correct with respect to the users’ needs in model-based development. During the development of a formal requirements model, the model must be extensively inspected and analyzed, as well as extensively tested and simulated.

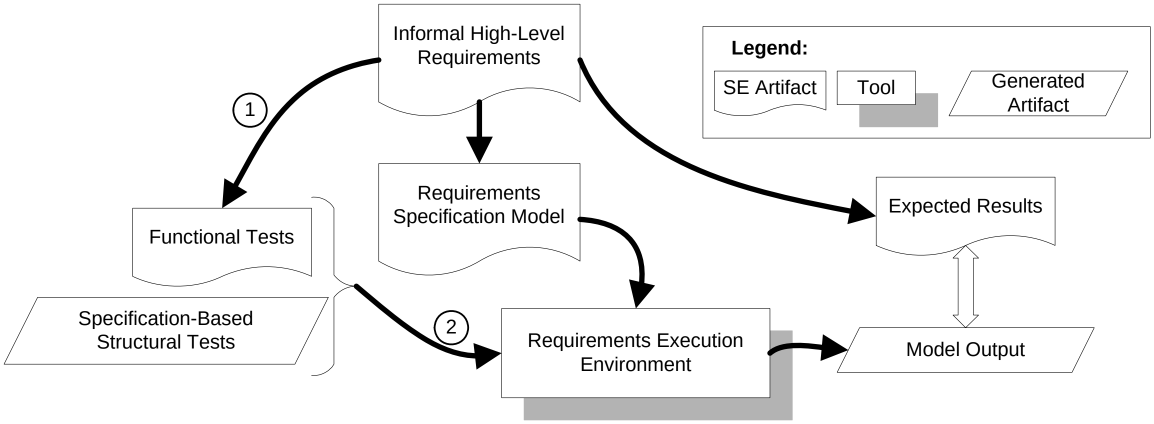
Informal High-Level Requirements
Legend:
SE Artifact
Tool
Generated Artifact
1
Requirements Specification Model
Expected Results
Functional Tests
Specification-Based Structural Tests
2
Requirements Execution Environment
Model Output

Figure 4: Validation testing in a model-based world.

A set of tests is developed from the informal high-level requirements to evaluate the required functionality of the model (step 1). The expected results from executing these tests are also manually derived. The tests used in this step must be developed based on the informal requirements and by domain experts; much like requirements-based tests in a traditional software development process. Currently, in both traditional and model-based development, the determination whether or not these tests adequately cover the informal requirements is based on engineering judgment—there is currently no objective notion of *requirements coverage*.

When used to test the model, these functional tests derived from the informal requirements may be considered adequate given some ad-hoc informal notion of requirements coverage but they may not, however, reach all the parts of the formal model—there may be transitions and conditions not exercised by these tests. The functional tests derived from the informal requirements will in most cases have to be complemented by a collection of model white-box tests developed specifically to exercise a model up to a certain level of *model coverage* (step 2 in Figure 4). Note here that these tests may be generated automatically helping us find the inputs that will take a model to hard-to-reach model elements, but we will still need a human oracle to determine if the result of each test is acceptable; that is, the test inputs used to cover the model could be automatically generated, but since we are testing the model for correctness with respect to the user's needs, the expected result must be derived from the informal requirements of acquired from domain experts.

When testing of the model has been completed and we are convinced that the model is correct, the testing process can switch from model validation testing to implementation conformance testing (Figure 5). In conformance testing, we are not interested in what the formal model does; we are interested in determining if the implementation correctly implements the formal model. Note here that the formal model now is the oracle; we assume the model is correct and all we want to show is that for all tests we run, the formal model provides the same result as the implementation. This is a radically different testing task than the validation testing discussed above.

All tests used during the validation testing of the formal specification can naturally be reused when testing the implementation (step 3 in Figure 5). The test-cases derived to test the formal specification provide the foundation for the testing of the implementation and the executable formal specification serves as an oracle during the testing of the implementation. This test set may, however, not provide adequate coverage of the implementation. For example, potential optimizations may have been used in the implementation that were not present in the model. Thus, the model-based tests will most likely have to be augmented with additional test-cases (step 4 in Figure 5).

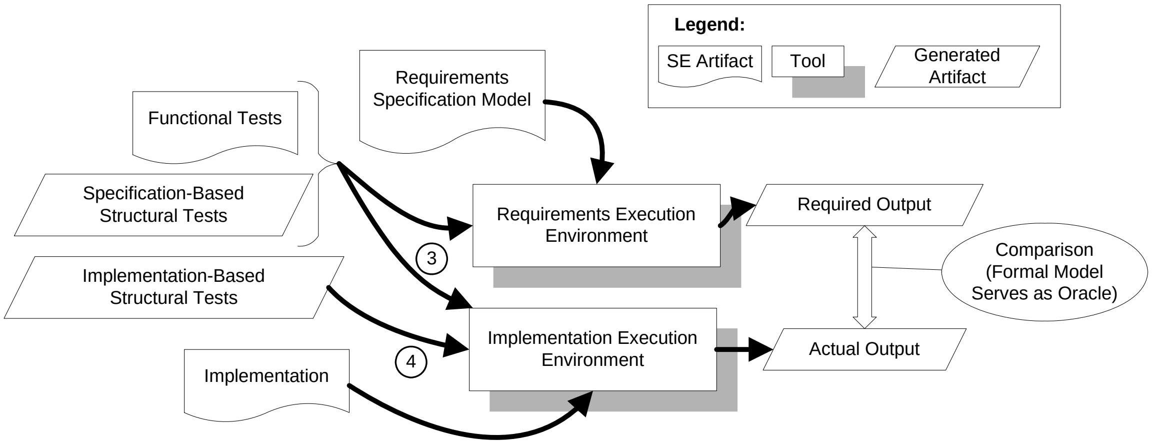
Legend:
SE Artifact
Tool
Generated Artifact
Requirements Specification Model
Functional Tests
Specification-Based Structural Tests
Implementation-Based Structural Tests
Implementation
Requirements Execution Environment
Implementation Execution Environment
3
4
Required Output
Actual Output
Comparison (Formal Model Serves as Oracle)

Figure 5: Conformance testing in a model-based world.

The characteristics of the two testing activities outlined above are radically different. Model validation testing currently requires largely manual development of the test-cases to ensure the model meets all the requirements and that no requirements have been missed when the model-was built (no errors of omission), and requires the manual development of the oracle. Conformance testing, on the other hand, could potentially be completely automated; since we assume the model is correct, tests can be generated from the model or implementation and the model can serve as an automated oracle.

The requirements on the test-cases used in the testing also differ between the two testing activities. In model validation testing we want our tests to represent clear and realistic operational scenarios; we do not want tests that require the aircraft to "fly upside-down, backwards, below sea level". In addition, we want tests that check related features to be similar in order to make test validation easier. When we perform conformance testing, on the other hand, we are simply testing if the implementation (or code generation) is correct and we are much less concerned about the characteristics of the test-cases; we simply want enough tests to convince us that the implementation is correct with respect to its specification.

The goal of our work described in this report is to (1) provide the foundation for *objective measure of the model-validation activities* so that we can move away from ad-hoc methods for determining requirements coverage, (2) provide the foundation for the *automation of requirements-based test-case generation* so that the high-cost of manually developing test-cases from the informal requirements can be reduced, and (3) provide *full automation of conformance testing.*

The remainder of this report will discuss these challenges in detail and point to some approaches that are promising and could be applied in production projects in the not too distant future. In addition, we identify a research agenda that should be addressed in future projects to reap the full benefits of model-based development.

1.3 How to Read This Report

We have structured this report to first provide an overview of the problem and the direction we advocate. We then provide more detailed descriptions of how these ideas can be put into practice and discuss case studies.

In Section 2 we clarify the relationship between high- and low-level requirements, formal models, and formal properties. In Section 3, we discuss the new challenges facing testing as we move to model-based development. In this section we provide a high-level view of the problems of requirements-based testing and point out promising directions. We also discuss the challenges with conformance-testing. In Section 4 we discuss in detail how the measurement of requirements coverage can be formalized. Autogenerating requirements-based test-cases for different notions of requirements coverage is discussed in Section 5. In Section 6 we discuss automated conformance testing and point out several challenges that must be addressed. Section 7 summarizes our work to date and provides direction for the future.

2 Requirements in a Model-Based World

Two documents govern the development of software requirements in the avionics industry. ARP 4754 [1] provides guidelines for the system-level processes and DO-178B [51] provides guidelines for the software development processes. The notion of *high-level software requirements* lie at the border between the systems engineering activities covered by ARP 4754 and the software engineering activities governed by DO-178B. Part of the system requirements are allocated to software, and *"high-level requirements are produced directly through analysis of system requirements"* (DO-178B, Section 5).

With respect to the discussion in this report, we can view the DO-178B software development process as composed of:

- **Requirements Gathering and Definition:** High-level requirements (HLR) are based on the systems requirements allocated to software and are defined during the software requirements process.
- **Software Architecture and Design:** Low-level requirements (LLR) and the software architecture are generally produced during the software design process.
- **Coding:** The software coding process produces the source code.
- **Integration:** The integration process produces object code and builds up to the integrated system or equipment.

Of particular interest in this report is the relationship between the high-level requirements and the low-level requirements. The definitions used in DO-178B are:

> **High-level requirements** - Software requirements developed from analysis of system requirements, safety-related requirements, and system architecture.
>
> **Low-level requirements** - Software requirements derived from high-level requirements, derived requirements, and design constraints from which source code can be directly implemented without further information.

The high-level software requirements (HLR) are produced directly through analysis of the system requirements and system architecture and their allocation to software. Lower-level software requirements are typically developed as part of the software development process. In fact, the DO-178B view of low-level requirements places them closer to a detailed design document than what in the broader software engineering community would be called a requirements document. As we move to model-based development, *we propose to replace the low-level requirements with the formal model*. The formal models of interest in the context of the Methods and Tools for Flight Critical Systems project would contain all information needed to proceed to coding; or, as advocated in model-based development, we can use code generation to replace the manual coding process. Nevertheless, when moving to a model-based world, the nature of requirements and the relationship between high-level requirements, low-level requirements, models, and properties has, in our opinion, not been adequately explored. Below we will present our view of how requirements, models, and formal properties fit together to form the basis for a requirements specification.

2.1 Requirements Framework

A formal model is used as a complement to the traditional requirements development effort. Before we define the relationship between the various artifacts in more detail, it is necessary to discuss the nature of formal models. An understanding of the pros and cons of different modeling styles is needed to better understand how we view the relationship between the artifacts developed in model-based development.

There are at least two well-known styles of formal specification. In a *constructive* (or model-based approach), one defines a model in terms of primitive types and constructors provided by the specification language. For example, models based on finite state machines (Stateflow [40], Statecharts [26], Simulink[39], SCADE [17], RSML^{-e}[54], SCR [31]) as well as notations based on sets and relations (Z [52], VDM [19]) fall in this category.

On the other hand, in *property based* (or axiomatic) specifications, one defines properties relating the operations or events of the system being specified without providing any information about the structure of the model itself. Of interest here, popular temporal logic specification techniques fall in this category, for example, computational tree logic (CTL) or linear time temporal logic (LTL) [14].

A disadvantage of a constructive style of specification is that it tends to bias the specifier and reader towards a particular implementation. No such bias exists in the property style of specification since no information is provided about the structure of the model being defined. An advantage of a constructive style of specification, however, is that it is used in common programming languages such as C and Ada and most engineers are immediately comfortable with it. A property-oriented specification is generally more difficult to understand and write. One also has to ensure that a property-oriented specification is *consistent* and *complete*. A specification is consistent if it always defines a single value for each operation on the same inputs (i.e., each operation is a mathematical *function*). A specification is complete if a result is specified for every set of inputs to an operation (i.e., each operation is a *total* function). Many constructive specification languages are designed so that *only* complete and consistent specifications can be written. In fact, the textbook method for showing that a property oriented specification is consistent is to create a constructive model of it and prove that all the properties in the property oriented specification hold over the constructive model. This establishes that at least one implementation of the specification exists and its properties must therefore be consistent.

The relationship between the various artifacts is illustrated in Figure 6. In model-based development as explored in the Methods and Tools for Flight Critical Systems project, the high-level requirements are used as the basis of the development of the model. We develop the model using a constructive modeling technique that yields an executable model that is usually easily accepted by practicing engineers. This model is developed to a level of detail where it could be used as a basis for manual coding or automated code generation without any additional information. In this sense, the model can be used to *replace* the low-level requirements as defined in DO-178B.

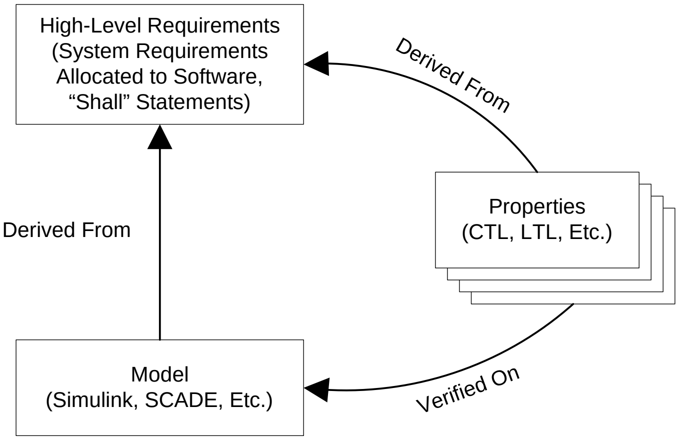
High-Level Requirements (System Requirements Allocated to Software, "Shall" Statements)
Derived From
Derived From
Properties (CTL, LTL, Etc.)
Model (Simulink, SCADE, Etc.)
Verified On

Figure 6: Relationship between high-level requirements, models, and properties.

The problem is now to determine if this model accurately captures the informal high-level requirements. The properties, captured using, for example, CTL, are very close in structure to the informal "shall" statements making up the high-level requirements; in fact, in [42] we argue that the informal "shall" requirements are simply desirable properties of our model expressed in an informal and ambiguous notation. Note here that there is not generally a one-to-one relationship between many of the informal high-level requirement and the properties. One requirement may lead to a number of properties. For example, consider the flight guidance system example used throughout this project. The flight guidance system example is described in detail in Appendix A and only a brief overview is included for clarity in this section.

A flight guidance system is a component of the overall Flight Control System (FCS). It compares the measured state of an aircraft (position, speed, and attitude) to the desired state and generates pitch and roll guidance commands to minimize the difference between the measured and desired state. The flight crew interacts with the FGS primarily through the Flight Control Panel (FCP). The FCP includes switches for turning the Flight Director (FD) on and off, switches for selecting the different flight modes such as vertical speed (VS), lateral navigation (NAV), heading select (HDG), altitude hold (ALT), lateral go around (LGA), approach (APPR), and lateral approach (LAPPR), the Vertical Speed/Pitch Wheel, and the autopilot disconnect bar. The FCP also supplies feedback to the crew, indicating selected modes by lighting lamps on either side of a selected mode's button.

Now, consider the requirement below taken from the flight guidance example.

"Only one lateral mode shall be active at any time."

This requirement leads to a collection of properties we would like our model to possess. The number of properties depends on which lateral modes are included in this particular configuration of the flight guidance system. The FGS used in our study has five lateral modes (ROLL, HDG, NAV, LGA, and LAPPR) leading to the following properties (stated informally).

1. *If ROLL is active, HDG, NAV, LGA, and LAPPR shall not be active.*
2. *If HDG is active, ROLL, NAV, LGA, and LAPPR shall not be active.*
3. *If NAV is active, ROLL, HDG, LGA, and LAPPR shall not be active.*
4. *If LGA is active, ROLL, HDG, NAV, and LAPPR shall not be active.*
5. *If LAPPR is active, ROLL, HDG, NAV, and LGA shall not be active.*

These properties can easily be formalized as LTL properties for verification in the model checker NuSMV.

```
1. G(Is_ROLL_Active-> (!Is_HDG_Active & !Is_NAV_Active &
   !Is_LGA_Active & !Is_LAPPR_Active))
2. G (Is_HDG_Active -> (!Is_ROLL_Active & !Is_NAV_Active &
   !Is_LGA_Active & !Is_LAPPR_Active))
3. G (Is_NAV_Active -> (!Is_ROLL_Active & !Is_HDG_Active &
   !Is_LGA_Active & !Is_LAPPR_Active))
4. G (Is_LGA_Active -> (!Is_ROLL_Active & !Is_HDG_Active &
   !Is_NAV_Active & !Is_LAPPR_Active))
5. G (Is_LAPPR_Active -> (!Is_ROLL_Active & !Is_HDG_Active &
   !Is_NAV_Active & !Is_LGA_Active))
```

In LTL, G means that the property is required to be true at all times (it is Globally the case that …).

On the other hand, some high-level requirements, for example, the requirement below (also from the FGS), lend themselves directly to a one-to-one formalization in the vocabulary of the model. (FD refers to the Flight Director and AP refers to the Auto Pilot.)

> *"If the onside FD cues are off, the onside FD cues shall be displayed when the AP is engaged."*

This can be formalized as shown in Property 1:

```
G((!Onside_FD_On & !Is_AP_Engaged)->
    X(Is_AP_Engaged -> Onside_FD_On))
```

Property 1: Formalization of FD cues requirement

The property states that it is globally true (G) that if the Onside FD is not on and the AP is not engaged, in the next instance in time (X) if the AP is engaged the Onside FD will also be on. Thus, capturing the informal high-level requirements as properties typically involves a certain level of refinement and extension to express the properties in the vocabulary of the formal model and to shore up any missing details to make formalization possible. For this reason, the formalization of the requirements is typically deferred until at least an early version of the model is available.

By formalizing these “shall” statements in a property specification language such as LTL we achieve several benefits. First, we can now use tools such as model checkers and theorem provers to verify that the properties (high-level requirements) indeed hold in the model (low-level requirements); an assurance that testing of the model alone cannot provide. Second, through the act of formalization and formal analysis using tools, we will most likely discover that many properties do not hold in the model; if that is the case, either the model or the property must be modified. In our experience, it is quite likely that the model is correct and the high-level requirement from which the property was developed was wrong or poorly written. Thus, the formalization and formal analysis provide valuable crosschecks of the informal-high-level requirements, the formal properties, and the model. Finally, with the formal properties derived from the high-level requirements and the model replacing the low-level requirements, we are better equipped to understand and explore the notion of tests derived from the high-level and low-level requirements; the main topic of this report that will be explored in detail in the Section 3.

2.2 Requirements Summary

To summarize, we view model-based development as a natural evolution of the traditional development process. The informal high-level software requirements are derived from the system requirements, safety requirements, and system architecture. The model is developed from these requirements and (hopefully) captures the behavior described in the high-level requirements. Any additional information needed for implementation not captured in the high-level requirements is captured in the model; the model replaces the low-level requirements.

The high-level requirements are formalized as properties that can be checked via simulations and requirements-based tests and proven against the model using automated tools such as model checkers and theorem provers. These activities will inevitably expose problems with the model, the high-level requirements, or the formalization of the properties that merit further investigation of the system under development and correction of the high-level requirements, the model, or the formalization of the properties (or any combination of the three) in an iterative fashion.

At the completion of this process we will have basically three artifacts; (1) the informal high-level requirements, (2) a model capturing the intended behavior of the system, and (3) a collection of formal properties derived from the high-level requirements (hopefully verified to hold in the model). We view this collection of artifacts to be the requirements specification of the system under development. Note here that it is quite possible that the size and complexity of the model could preclude the verification that the properties hold in the model; completely automated analysis techniques, such as model checking, have limited scalability. Manually-assisted techniques, such as theorem proving, require a significant amount of human expertise and time to use on large systems and may be impractical. The testing approaches discussed in the remainder of this report are applicable regardless of whether the properties have been verified in the model or not.

3 Testing in a Model-Based World

As mentioned in the introduction, the move to model-based development and the new relationship between high-level requirements, low-level requirements (the model), and properties lead to new challenges and opportunities in the testing of software. Currently, DO-178B requires that we demonstrate (among many other things) the objectives in Table 1.

1. The executable object code complies with the high-level requirements [DO-178B, Table A-6, Item 1]
2. The executable object code complies with the low-level requirements [DO-178B, Table A-6, Item 3]
3. Test coverage of high-level requirements is achieved [DO-178B, Table A-7, Item 3]
4. Test coverage of low-level requirements is achieved [DO-178B, Table A-7, Item 4]
5. Test coverage of [the implementation] is achieved [DO-178B, Table A-7, Items 5-8]

Table 1: Test objectives as defined in DO-178B, Table A-7.

To satisfy these objectives, test-cases are derived from the high-level requirements (let us call these High-level Requirements-based Tests or HRT) as well as the low-level requirements (Low-level Requirements-based Tests or LRT). To our knowledge, there is no objective measure of *"requirements coverage"* to meet bullets 3 and 4. Instead, coverage is demonstrated through a best effort using traditional techniques such as partition analysis, the category partition method, and boundary value analysis. These tests represent realistic operational scenarios and are extensively validated by domain-experts.

When we have the code in hand, we are in a position to objectively measure the code coverage achieved when executing the HRT and LRT. Should this coverage be too low, we must either (1) develop more requirements-based test-cases to cover the uncovered code, (2) extend the requirements in case the uncovered code reveals desirable behavior not covered in the requirements, (3) determine that the code is valid but has been deactivated, or (4) declare the code superfluous (dead) and remove it from the implementation. With the completion of this step we have demonstrated test coverage of the implementation (bullet 5). When the tests are executed on the executable object code running on the target platform, for certification purposes, correct execution of the tests adequately demonstrates that the object executable code complies with the high-level and low-level requirements (bullets 1 and 2).

As the role of traditional requirements changes as described in Section 2.1 and the use of code generation from a model becomes prevalent, the role of high- and low-level requirements in the testing process changes in that "low-level requirements" will be replaced with "model".

The process of assuring that the implementation accurately reflects the requirements is now a two-step testing process. First, we want to assure that the model accurately reflects the high-level requirements. Second, we want to demonstrate that the source code derived from the model (whether manually or through a code generator) exhibits the same behavior as the model.

As defined in Section 1.2 and above, the nature of these two testing activities is radically different. The *validation testing* of the model is done with requirements-based test sets derived from the high-level requirements in collaboration with domain experts. These tests should represent realistic usage scenarios of the system and the expected outcomes of the tests are derived from the high-level requirements or through consultation with domain experts. These tests are used to determine that we have the right high-level requirements and that the model accurately captures the high-level requirements.

The tests developed to demonstrate that the implementation conforms to the model can be of a completely difference nature. Now we are interested in *conformance testing* where we are no longer concerned with the behavior of the model and code, we simply want to show that they have the *same* behavior. These tests need to provide some coverage of the model and implementation to assure that they conform to each other; for example, we may want MC/DC coverage of the model as well as the code. The tests do not, however, have to represent realistic scenarios since they are not intended for validation.

From the discussion above, there are several issues that merit further scrutiny. First, given the close relationship between the informal high-level requirements and the property specifications, can the properties be used to somehow help us objectively measure how well a suite of test-cases exercises the high-level requirements? Second, can the high-level requirements and the properties be used to help automatically generate requirements-based tests? Third, since we aim to exercise the model up to a certain level of coverage, what would a suitable collection of coverage criteria be? Finally, can the model be effectively used to generate conformance tests for the implementation? What generation technique works well? What coverage criteria are suitable in these generation tasks? Should such conformance tests supplement the requirements-based tests or be generated in addition to the requirements-based tests? In the remainder of this report we will address both requirements-based testing and model-based testing.

4 Requirements-Based Tests — Validation Testing

Recent developments in test-case generation techniques have made it possible to generate vast numbers of test-cases from a formal model [3],[30],[21],[43],[44],[46],[47],[49]. Unfortunately, generating tests from the formal model and using them to test the formal model tells us little if anything about the correctness of the model. This would be an activity analogous to white-box testing of an implementation. This approach to testing will not expose errors of omission and is designed to exercise the structure of the model (or implementation) rather than demonstrating compliance with the high-level required behavior of the system. In particular, the model cannot be used as an oracle to determine if a test has the desired outcome. The decision if a test yields the desired outcome must be made with respect to the high-level software requirements.

In this section we address the notion of requirements-based tests—tests derived from the requirements and intended to exercise the requirements up to some level of requirements-based test adequacy coverage.

4.1 High-Level Requirements Coverage

As mentioned in Section 2.1, there is a close relationship between the high-level requirements and the properties captured for verification purposes. As an example, consider the requirement from the sample FGS we briefly discussed earlier (and in Appendix A).

> *"If the onside FD cues are off, the onside FD cues shall be displayed when the AP is engaged."*

In our work we translated this requirement to a formula in Linear Time Temporal Logic (LTL) and verified that our model of the FGS satisfied this requirement using a model checker (NuSMV). The LTL property is very similar in structure to the natural language requirements and this translation was quite easy to perform manually.

```
G((!Onside_FD_On & !Is_AP_Engaged)->
    X(Is_AP_Engaged -> Onside_FD_On))
```

In our previous experience, practicing engineers have no problems developing such formal properties.

An analyst developing test-cases from the informal requirements might derive the scenario in Table 2 to demonstrate that the requirement is met.

1. Turn the Onside FD off
2. Disengage the AP
3. Engage the AP
4. Verify that the Onside FD comes on

Table 2: Manually developed requirements-based test scenario.

Does this adequately cover this high-level requirement? Does passing such a test-case indicate that the model correctly has captured the behavior required through this requirement? If not, what would additional test-cases look like? The specification of the requirement as a property allows us to define several objective criteria with which to determine whether we have adequately tested the requirement. We hypothesize that coverage of such criteria can serve as a reliable measure of the thoroughness of the requirements-bases testing and model-validation activities.

Several different structural coverage criteria that have been investigated for source code and for executable modeling languages, for example, [49], [45], [43], and [21]. It is possible to adapt some of these criteria to fit in the domain of high-level requirements captured as formal properties in, for example, CTL or LTL. For instance, consider the notion of decision coverage of source code where we are required to find two test-cases (one that makes the decision true and another one that makes the decision false). If we adapt this coverage criterion to apply to high-level requirements it would require a single test-case that demonstrates that the requirement is satisfied by the model—a test-case such as the manually developed one in Table 2. If we derived such a test-case for each requirement, we could claim that we have achieved requirements decision coverage.

This example brings up a distinction between structural coverage metrics over source code and coverage metrics over requirements. Metrics over code assume that a Boolean expression can take on both 'true' and 'false' values. When generating tests from requirements, we usually are interested in test-cases exercising the different ways of satisfying a requirement (i.e., showing that it is true). Test-cases that presume the requirement is 'false' are not particularly interesting; this is discussed more in Section 4.2.2.

Recall that the high-level requirements are naturally expressed as formal properties; the concept of structural coverage of requirements expressed as formal properties can now naturally be extended to address more demanding coverage criteria, for example, MC/DC. The notion of structural coverage over property syntax will be discussed in detail in Section 4.2.

Although we in this report focus on requirements specified in temporal logic there are other notations that can be used to describe high-level requirements. For example, SCADE [17] and Reactis [48] use *synchronous observers* [24], which are small specifications of high-level requirements written as state machines or in the same notation as the software specification that run "in parallel" with the model. The observer emits one Boolean output, and the property is true if this output is always true. We briefly discuss coverage of requirements expressed as synchronous observer in Section 4.3.

4.2 Structural Coverage over LTL Syntax

As mentioned above, structural coverage criteria defined for source code and for executable modeling languages can be adapted to fit propositional temporal logics such as CTL and LTL.

For instance, decision coverage of the requirements would require a single test-case that demonstrates that the requirement is satisfied by the model—a test-case such as the manually developed one in Table 2. Typically, this single positive test-case is too weak of a coverage since it is often possible to derive a vacuous test-case. This vacuity is a serious problem when test-cases are automatically generated and not manually inspected. If we again consider our sample requirement on the FGS

> *"If the onside FD cues are off, the onside FD cues shall be displayed when the AP is engaged."*

formalized as

```
G((!Onside_FD_On & !Is_AP_Engaged)->
    X(Is_AP_Engaged -> Onside_FD_On))
```

Given this requirement we can satisfy the decision coverage metric by creating a test-case that leaves the autopilot disengaged throughout the test and disregard the behavior of the flight director. Although this test-case technically satisfies the property, it does not shed much light on the correctness of our model.

A better alternative would be to adapt one of the more rigorous structural coverage criteria used in the avionics software domain, such as, the Modified Condition/Decision Coverage (MC/DC) criterion, for use in the requirements-based testing domain. In this section, we define a requirements coverage metric called *Unique First Cause Coverage* (UFC) that is adapted from MC/DC criteria.

The MC/DC metric was developed to meet the need for extensive testing of complex Boolean expressions in safety-critical applications [12]. MC/DC was developed as a practical and reasonable compromise between decision coverage and multiple condition coverage. It has been in use for several years in the commercial avionics industry. The important aspect of this criterion is the requirement that testing should demonstrate the independent effect of atomic Boolean conditions on the Boolean expressions in which they occur.

A test-suite is said to satisfy MC/DC if executing the test-cases in the test-suite will guarantee that:

- every point of entry and exit in the model has been invoked at least once,
- every basic condition in a decision in the model has taken on all possible outcomes at least once, and
- each basic condition has been shown to independently affect the decision's outcome

where a basic condition is an atomic Boolean valued expression that cannot be broken into Boolean sub-expressions and a decision is a Boolean expression appearing somewhere within a model or program.

There are multiple forms or templates that are used for ensuring the independence of conditions. In this report, we use *masking* MC/DC [27] to determine independence. In masking MC/DC, a basic condition is *masked* if varying its value cannot affect the outcome of a decision due to structure of the decision and the value of other conditions. To satisfy masking MC/DC for a condition, we must have test states in which the condition is not masked (i.e., changes the outcome of the decision) and takes on both 'true' and 'false' values. Test-case pairs for different basic conditions need not necessarily be disjoint. In fact, the size of MC/DC adequate test-suite can be as small as *N+1* for a decision point with *N* conditions.

4.2.1 MC/DC Coverage of Decisions

The masking criteria are defined over the Boolean operators (and, for imperative programs, loops and selection statements). For example, given the expression *A **and** B*, to show the independence of *B*, we must hold the value of *A* to true; otherwise varying *B* will not affect the outcome of the expression. We define the test scenarios necessary to exercise the basic expressions in Table 3.

Expression	**Required assignments to *(A, B)***
A **and** B	{ (T, T), (T, F), (F, T) }
A **or** B	{ (T, F), (F, T), (F, F) }
not A	{T, F}

Table 3: Necessary Assignments to *A* and *B* to satisfy MC/DC on basic gates.

When we consider decisions with multiple Boolean operators, we must ensure that the test results for one operator are not masked out by the behavior of other operators. For example, given *A **or** (B **and** C)* the tests for *B **and** C* will not affect the outcome of the decision if *A* is true.

It is straightforward to describe the set of required MC/DC assignments for a decision as a set of Boolean expressions. Each expression is designed to show whether a particular condition positively or negatively affects the outcome of a decision. That is, if the expression is true, then the corresponding condition is guaranteed to affect the outcome of the decision. Given a decision *A*, we define A^+ to be the set of expressions necessary to show that all of the conditions in *A positively* affect the outcome of *A*, and A^- to be the set of expressions necessary to show that the all of the conditions in *A negatively* affect the outcome of *A*.

We can define A^+ and A^- schematically over the structure of complex decisions as follows:

$x^+ = \{x\}$ (where x is a condition)

$x^- = \{\textbf{not}\ x\}$ (where x is a condition)

The positive and negative test-cases for conditions are simply the singleton sets containing the condition and its negation, respectively.

$(A\ \textbf{and}\ B)^+ = \{a\ \textbf{and}\ B \mid a \in A^+\} \cup \{A\ \textbf{and}\ b \mid b \in B^+\}$

The formula states that to get positive MC/DC coverage over *A* **and** *B*, we need to make sure that every element in A^+ uniquely contributes to making *A* **and** *B* true while holding *B* true, and similarly we also have to ensure that every element in B^+ contributes to making *A* **and** *B* true while holding *A* true.

$(A\ \textbf{and}\ B)^- = \{a\ \textbf{and}\ B \mid a \in A^-\} \cup \{A\ \textbf{and}\ b \mid b \in B^-\}$

Negative MC/DC coverage over *A* **and** *B* is analogous to positive MC/DC coverage. The only difference is that here we have to ensure that every element in A^- and every element in B^- uniquely contributes to making *A* **and** *B* false.

$(A\ \textbf{or}\ B)^+ = \{a\ \textbf{or}\ \textbf{not}\ B \mid a \in A^+\} \cup \{\textbf{not}\ A\ \textbf{or}\ b \mid b \in B^+\}$

To make *A* ***or*** *B* true, either only *A* or only *B* have to be true or both can be true. For positive MC/DC coverage, we only need to consider the cases where either only *A* or only *B* are true. Therefore we would have to make sure that every element in A^+ is uniquely contributing to making *A* **or** *B* true while formula B is false, and similarly every element in B^+ is uniquely contributing to making the formula true while *A* is false.

$(A\ \textbf{or}\ B)^- = \{a\ \textbf{and not}\ B \mid a \in A^-\} \cup \{\textbf{not}\ A\ \textbf{and}\ b \mid b \in B^-\}$

Negative MC/DC coverage considers the case when *A* and *B* are false, and ensures that every element in A^- and every element in B^- uniquely contributes to making *A* **or** *B* false.

$(\textbf{not}\ A)^+ = \{\mathrm{a} \mid \mathrm{a} \in A^-\} = A^-$

$(\textbf{not}\ A)^- = \{\mathrm{a} \mid \mathrm{a} \in A^+\} = A^+$

The positive and negative MC/DC coverage sets for **not** *A* would be inverse of what they were for *A*.

Each of the expressions in the positive and negative sets can be seen as defining a constraint over a program or model state. The process of satisfying MC/DC involves determining whether each of these constraints is satisfied by some state that is reached by a test within a test-suite.

4.2.2 Unique First Cause (UFC) Coverage over Paths

There are important distinctions between testing of models and testing of properties. For an LTL property, the truth or falsehood of the property is determined over an infinite length path, whereas a test-case is of finite length. When defining structural coverage over a property, we wish to determine whether the tests in a set are *prefixes* of infinite satisfying paths that exercise the structure of the property in certain ways. Thus, to perform requirements coverage, we would like to apply the same structural coverage ideas to LTL properties rather than models or programs by extending this notion to apply to paths. The idea is to measure whether we have sufficient tests to show that all atomic conditions within the property affect the outcome of the property. Equivalently, we would like to show that for every atomic condition within a formula, we have a trace prefix that contains a state in which that atomic condition is necessary for the formula to be true.

Since requirements captured as LTL properties define paths rather than states, we must broaden our view of structural coverage to accommodate satisfying paths rather than satisfying states. We can define these paths by extending the constraints for state-based MC/DC to include temporal operators. These operators describe the path constraints required to reach an acceptable state. The idea is to characterize a trace σ in which the formula holds for states $\sigma(0)...\sigma(k-1)$, then passes through state $\sigma(k)$, in which the truth or falsehood of the formula is determined by the atomic condition of interest. For satisfying traces, we require that the formula continue to hold thereafter.

A test-suite is said to satisfy Unique First Cause (UFC) over a set of LTL formulas if executing the test-cases in the test-suite will guarantee that:

- every basic condition in a formula has taken on all possible outcomes at least once
- each basic condition has been shown to independently affect the formula's outcome

We define independence in terms of the shortest satisfying path for the formula. Thus, if we have a formula A and a path σ, an atom a in A is the unique first cause if, in the first state along σ in which A is satisfied, it is satisfied because of atom a. To make this notion concrete, suppose we have the formula $\mathrm{F}(a \vee b)$ and a path σ in which a was initially true in step $\sigma(2)$ and b was initially true in step $\sigma(5)$. For path σ, a (but not b) would satisfy the unique first cause obligation.

The definition above takes the evenhanded view that all formulas will be *satisfiable,* that is, there are some traces in which the property is true and some in which it is false. Nevertheless, we eventually want a system in which all of the properties (requirements) are *valid*. Therefore, we are primarily concerned with the *positive* set of UFC test-cases for a formula. For completeness and ease of definition, we define both the positive and negative constraints sets for each LTL operator.

For the formalization of UFC coverage of requirements expressed as LTL properties we will use the notational conventions that were defined above for Boolean expressions and extend them to include temporal operators in LTL. The meaning of the basic LTL operators is defined as in Table 4.

Operator	Notation	Meaning
Globally *A*	G(*A*)	Formula *A* is true in all states
Future *A*	F(*A*)	Formula *A* is true in some future state
A until *B*	*A* U *B*	Formula *A* is true in every state until *B* becomes true. *B* must eventually become true for the property to be true.

Table 4: Meaning of LTL Operators

We now extend A^+ and A^- defined over states to define satisfying paths over LTL temporal operators as follows:

$\mathrm{G}(A)^+ = \{A \text{ U } (a \textbf{ and } \mathrm{G}(A)) \mid a \in A^+\}$

G(*A*) is true if *A* is true along all states within a path. The *A* U (*a* **and** G(*A*)) formula ensures that each element *a* in A^+ contributes to making *A* true at some state along a path in which *A* is globally true.

$\mathrm{G}(A)^- = \{A \text{ U } a \mid a \in A^-\}$

G(*A*) is falsified if there is some state along the path in which *A* is false. Each F *a* formula demonstrates that G(A) is false because of a particular condition $a \in A^-$.

$\mathrm{F}(A)^+ = \{ (\textbf{not } A) \text{ U } a \mid a \in A^+\}$

The unique effect of $a \in A^+$ for the F(A) formula is demonstrated by showing that it uniquely contributes to making *A* true in some future state.

$\mathrm{F}(A)^- = \{\textbf{not } A \text{ U } (a \textbf{ and } \mathrm{G}(\textbf{not } A)) \mid a \in A^-\}$

F(*A*) is false if there is no state in which *A* holds. The (**not** *A* U *(a* **and** G(**not** *A*))) formula ensures that *A* is globally false and demonstrates the unique effect of $a \in A^-$ by showing that there exists some state along the path in which *A* is false because of *a*.

$(A \text{ U } B)^+ = \{(A \textbf{ and not } B) \text{ U } ((a \textbf{ and not } B) \textbf{ and } (A \text{ U } B)) \mid a \in A^+\} \cup$
$\{(A \textbf{ and not } B) \text{ U } b \mid b \in B^+\}$

For the formula *A* U *B* to hold, *A* must hold in all states until we reach a state where *B* holds. Therefore, positive UFC coverage for this would mean we have to ensure that every element in A^+ contributes to making *A* true along the path and every element in B^+ contributes to making the formula true.

The formula to the left of the union provides positive UFC over *A* in (*A* U *B*). Recall that an ‘until’ formula is immediately satisfied if *B* holds. Therefore, in order to show that some specific atom in *A* (described by *a*) affects the outcome, we need to show that this atom is necessary *before B* holds. This is accomplished by describing the prefix of the path in which *a* affects the outcome as: (*A* **and not** *B*) U ((*a* **and not** *B*) …). In order to ensure that our prefix is sound, we still want *B* to eventually hold, we add (*A* U *B*) to complete the formula.

The formula on the right of the union is similar and provides positive UFC over *B* in *A* U *B*. In this case we want to ensure that *B does not hold* on the left of the 'until' to ensure that *b* is the reason that *A* U *B* holds along this path.

$(A \text{ U } B)^{-} = \{(A \textbf{ and not } B) \text{ U } (a \textbf{ and not } B) \mid a \in A^{-}\} \cup$
$\{(A \textbf{ and not } B) \text{ U } (b \textbf{ and not } (A \text{ U } B)) \mid b \in B^{-}\}$

For the formula *A* U *B* to be falsified, *A* does hold until a state in which *B* is true. The formula to the left of the union demonstrates that $a \in A^{-}$ uniquely contributes to the falsehood of *A* by describing a path in which *A* holds (and *B* does not – otherwise the formula *A* U *B* would be true) until a state in which *a* falsifies *A* and *B* still does not hold. The formula to the right of the union describes the situation where $b \in B^{-}$ prevents *B* from holding along some state on a path in which *A* U *B* does not hold thereafter.

$\text{X}(A)^{+} = \{\text{X}(a) \mid a \in A^{+}\}$

Every test-case in A^{+} contributes to *A* holding in the next state.

$\text{X}(A)^{-} = \{\text{X}(a) \mid a \in A^{-}\}$

Every test-case in A^{-} contributes to *A* not holding in the next state.

The definitions above define a notion of requirements UFC coverage over execution traces. Since we have been working with linear time temporal logic, the definitions apply over infinite traces. Naturally, test-cases must by necessity be finite; therefore, the notion of requirements coverage must be adopted to apply to finite traces. This is the topic of the next section.

4.2.3 Adapting Formulas to Finite Tests

We would like to use the LTL property formulations to measure whether a set of test-cases adequately tests requirements. LTL is normally formulated over *infinite* paths, however, while test-cases correspond to *finite* paths. Nevertheless, the notion of coverage defined in the previous section can be straightforwardly adapted to consider finite paths as well. There is a growing body of research in using LTL formulas as monitors during testing [38], [15], [5], and we can adapt these ideas to check whether a test-suite has sufficiently covered a property.

Manna and Pnueli ([38]) define LTL over *incomplete* models, that is, models in which some states do not have successor states. In this work, the operators are given a best-effort semantics, that is, a formula holds if all evidence along the finite path supports the truth of the formula. One consequence of this formulation is that the next operator (X) is split into two operators: X and X!, which are *strong* and *weak* next state operators, defined (informally) in Table 5:

Operator	Notation	Meaning
Globally *A*	G(*A*)	Formula *A* is true in all states along the finite path
Future *A*	F(*A*)	Formula *A* is true in some future state along the finite path
A until *B*	*A* U *B*	Formula *A* is true in every state until *B* becomes true. *B* must eventually become true along the finite path for the property to be true.
Weak next *A*	X(*A*)	True if either 1.) we are in the last state of the finite path, or 2.) *A* is true in the next state
Strong next *A*	X!(*A*)	True if both 1.) we are not in the last state of the finite path and 2.) *A* is true in the next state
A weak-until *B*	*A* W *B*	Derived operator: (*A* W *B*) ⇔ ((*A* U *B*) **or** G(A)). True if formula *A* is true in every state until *B* is true. This formula does not require that *B* is eventually true.

Table 5: Informal Descriptions of Finite LTL Operators

It is straightforward to define a formal semantics over finite paths. We assume that a state is a labeling $L : V \rightarrow \{T, F\}$ for a finite set of variables *V*, and that a finite path π of length *k* is a sequence of states $s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_k$. We write π^i for the suffix of π starting with state s_i, and the length of the path as $|\pi|$ Given these definitions, the formal semantics of LTL over finite paths is defined in Table 6.

1. $\pi \models$ **true**
2. $\pi \models p$ iff $|\pi| > 0$ and $p \in L(s_1)$
3. $\pi \models$ **not** A iff $\pi \not\models A$
4. $\pi \models$ (A **and** B) iff $\pi \models A$ and $\pi \models B$
5. $\pi \models$ (A **or** B) iff $\pi \models A$ or $\pi \models B$
6. $\pi \models X(A)$ iff $|\pi| \leq 1$ or $\pi^2 \models A$
7. $\pi \models X!(A)$ iff $|\pi| > 1$ and $\pi^2 \models A$
8. $\pi \models G(A)$ iff for all $1 \leq i \leq |\pi|$, $\pi^i \models A$
9. $\pi \models F(A)$ iff for some $1 \leq i \leq |\pi|$, $\pi^i \models A$
10. $\pi \models A \text{ U } B$ iff there is some $1 \leq i \leq |\pi|$ where $\pi^i \models B$ and for all $j = 1..(i\text{-}1)$, $\pi^j \models A$

Table 6: Semantics of LTL over Finite Paths

As expected, these definitions correspond with the standard semantics except that they do not require that *G* properties hold infinitely (only over the length of the finite path), and do not require *X* properties to hold in the last state of a finite path.

The semantics in Table 6 are sensible and easy to understand, but may be too strong for measuring test coverage. We may want to consider tests that show the independence of one of the atoms even if they are "too short" to discharge all of the temporal logic obligations for the original property. For example, consider the formula:

((a **or** b) U c)

and the test-cases in Table 7:

Test 1:

Atom	Step 1	Step 2	Step 3
a	t	t	f
b	t	f	t
c	f	f	f

Test 2:

Atom	Step 1	Step 2
a	t	t
b	t	f
c	f	t

Table 7: Test-suite for Property ((*a* or *b*) U *c*)

Are these two test-cases sufficient to show the independent effects of *a, b,* and c? From one perspective, test 1 is (potentially) a prefix of a path that satisfies ((a **or** b) U c) and independently shows that *a* and *b* affect the outcome of the formula—the test-case illustrates that the formula holds with only *a* or only *b* being true. Test 2 shows the independent effect of *c*. From another perspective (the perspective of the finite semantics described above), test 1 does not satisfy the formula (since in the finite semantics in Table 6 requires that for the until formula to hold *c* must become true in the path), so cannot be used to show the independent effect of any of the atoms.

The issue with these tests (and with finite paths in general) is that there may be *doubt* as to whether the property as a whole will hold. This issue is explored in [15], which defines three different semantics for temporal operators: *weak, neutral,* and *strong.* The *neutral* semantics are the semantics of [38] described in the previous section (Table 5 and Table 6). The *weak* semantics do not require eventualities (*F* and the right side of *U*) to hold along a finite path, and so describe prefixes of paths that may satisfy the formula as a whole. The *strong* semantics always fail on *G* operators, and therefore disallow finite paths if there is any doubt as to whether the stated formula is satisfied.

To allow test-suites like the one in Table 7 to satisfy the independence obligations, we slightly weaken our LTL obligations. In this case, given a formula *f*, we are interested in a prefix of an accepting path for *f* that is long enough to demonstrate the independence of our condition of interest. Thus, we want the operators leading to this demonstration state to be *neutral*[1], but the operators afterwards can be *weak.*

The *strong* and *weak* semantics are a coupled dual pair because the negation operator switches between them. In [15], the semantics are provided as variant re-formulations of the neutral semantics. However, they can also be described as syntactic transformations of neutral formulas that can then be checked using the neutral semantics. The representations are equivalent, and we refer the reader to [15] for a full description of the three semantics and their effect on provability within the defined logic.

We define *weak* [*F*] to be the weakening of a formula *F* and *strong* [*F*] to be the strengthening of formula *F*. The transformations *weak* and *strong* are defined in Table 8.

1. *weak*[**true**] = **true**
2. *weak*[*p*] = *p*
3. *weak*[**not** *A*] = **not** *strong*[*A*]
4. *weak*[*A* **and** *B*] = *weak*[*A*] **and** *weak*[*B*]
5. *weak*[*A* **or** *B*] = *weak*[*A*] **or** *weak*[*B*]
6. *weak*[X!(*A*)] = X(*weak*[*A*])
7. *weak*[X(*A*)] = X(*weak*[*A*])
8. *weak*[G(*A*)] = G(*weak*[*A*])
9. *weak*[F(*A*)] = **true**
10. *weak*[A *U* B] = *weak*[*A*] W *weak*[B]
11. *weak*[A *W* B] = *weak*[*A*] W *weak*[B]
12. *strong*[**true**] = **true**
13. *strong*[*p*] = *p*
14. *strong*[**not** *A*] = **not** *weak*[*A*]
15. *strong*[*A* **and** *B*] = *strong*[*A*] **and** *strong*[*B*]
16. *strong*[*A* **or** *B*] = *strong*[*A*] **or** *strong*[*B*]
17. *strong*[X!(*A*)] = X!(*strong*[*A*])
18. *strong*[X(*A*)] = X!(*strong*[*A*])
19. *strong*[G(*A*)] = **false**
20. *strong*[F(*A*)] = F(*strong*[*A*])
21. *strong*[A *U* B] = *strong*[*A*] U *strong*[B]
22. *strong*[A *W* B] = *strong*[*A*] U *strong*[B]

Table 8: Definitions of *weak* and *strong* LTL transformations

Given these transformations, we can re-formulate the necessary UFC paths in LTL. The idea is that we want a prefix of a satisfying path that conclusively demonstrates that a particular condition affects the outcome of the formula. To create such a prefix, we want a *neutral* formula up to the state that demonstrates the atomic condition and a *weak* formula thereafter. The modified formulas defining requiring UFC over finite prefixes are shown in Table 9.

[1] The *strong* semantics are too strong – any property containing a G-operator will be disproved.

$$G(A)^{+} = \{A \mathrm{\ U\ } (a \mathbf{\ and\ } \mathit{weak}[G(A)]) \mid a \in A^{+}\}$$

$$G(A)^{-} = \{\, A \mathrm{\ U\ } a \mid a \in A^{-}\}$$

$$F(A)^{+} = \{(\mathbf{not\ } A) \mathrm{\ U\ } a \mid a \in A^{+}\}$$

$$F(A)^{\,-} = \{\mathbf{not\ } A\ U\ (a \mathbf{\ and\ } \mathit{weak}[G(\mathbf{not\ } A)])\mid a \in A^{-}\}$$

$$\begin{aligned}(A \mathrm{\ U\ } B)^{+} = \{&(A \mathbf{\ and\ not\ } B) \mathrm{\ U\ } ((a \mathbf{\ and\ not\ } B) \mathbf{\ and\ } \mathit{weak}[(A \mathrm{\ U\ } B)]) \mid a \in A^{+}\} \cup \\ \{&(A \mathbf{\ and\ not\ } B) \mathrm{\ U\ } b \mid b \in B^{+}\}\end{aligned}$$

$$\begin{aligned}(A \mathrm{\ U\ } B)^{-} = \{&(A \mathbf{\ and\ not\ } B) \mathrm{\ U\ } (a \mathbf{\ and\ not\ } B) \mid a \in A^{-}\} \cup \\ \{&(A \mathbf{\ and\ not\ } B) \mathrm{\ U\ } (b \mathbf{\ and\ not\ } \mathit{weak}[(A \mathrm{\ U\ } B)]) \mid b \in B^{-}\}\end{aligned}$$

$$X(A)^{\,+} = \{X(a) \mid a \in A^{+}\}$$

$$X(A)^{\,-} = \{\, X(a) \mid a \in A^{-}\}$$

Table 9: Weakened UFC LTL Formulas describing Accepting Prefixes

The only formulas that are changed in Table 9 from the original formulation in Section 4.2.2 are $G(A)^{+}$, $F(A)^{\,-}$, one branch of $(A \mathrm{\ U\ } B)^{+}$, and one branch of $(A \mathrm{\ U\ } B)^{-}$. These are the formulas that have additional obligations to match a prefix of an accepting path after showing how the focus condition affects the path.

4.2.4 Discussion on Structural Coverage over Property Syntax

It is possible to take two possible views when measuring the coverage of requirements from a given test-suite. The first perspective states that each test-case must have sufficient evidence to demonstrate that the formula of interest is true and that a condition of interest affects the outcome of the formula. This perspective can be achieved using the *neutral* finite LTL rules and our original property formulation.

The second perspective states that each test-case is a *prefix* of an accepting path for the formula of interest and that the condition of interest affects the outcome of the formula. This perspective can be achieved using the weakened UFC obligations shown in Table 9.

Making this discussion concrete, given our example formula:

$$f = ((a \mathbf{\ or\ } b) \mathrm{\ U\ } c)$$

the UFC obligations for the original and modified rules are shown in Table 10.

{((*a* **or** *b*) **and not** *c*) U ((*a* **and not** *c*) **and** *f*),
((*a* **or** *b*) **and not** *c*) U ((*b* **and not** *c*) **and** *f*),
(*a* **or** *b*) **and not** *c*) U *c* }

(1)

{((*a* **or** *b*) **and not** *c*) U ((*a* **and not** *c*) **and** ((*a* **or** *b*) W *c*)),
((*a* **or** *b*) **and not** *c*) U ((*b* **and not** *c*) **and** ((*a* **or** *b*) W *c*)),
(*a* **or** *b*) **and not** *c*) U *c* }

(2)

Table 10: UFC obligations for ((*a* or *b*) U *c*) in (1) original formulation and (2) weakened formulation

In the original formulation (1), the first two obligations are not satisfied by the test-suite in Table 7 because *c* never becomes true in the first test-case. In (2), however, the requirement is covered, because the first test-case is a potential prefix of an accepting path. Further research is required to determine which formulation is best for deciding requirements coverage. In particular, we are interested in determining which formulation will give us adequate coverage of the model under investigation and the fault finding potential of the test-suited derived to provide the desired coverage.

4.3 Structural Coverage Criteria over Synchronous Observers

As mentioned in the introduction to this section, an alternative to capturing the requirements of a system as formal properties we can instead capture the required behavior as a small model in some other notation, for example, finite state machines, Simulink, or Stateflow. These requirements-models are often called *synchronous observers* [24] since they can be viewed as existing in parallel with the model of interest observing the behavior of that model (Figure 7); should the model exhibit undesirable behavior, the observer will enter some dedicated error state. In this way, the check that a required property holds can be reformulated from an LTL formula to a simple reachability property of the observer ("the observer will never enter the error state").

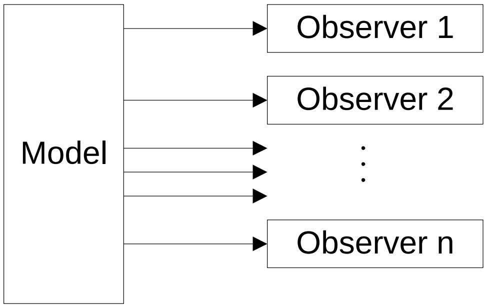

Figure 7: Synchronous observers monitoring a model's behavior.

Naturally, the notion of requirements coverage can now be viewed as structural coverage over the observer. Below we will discuss how requirements can be formulated in two different ways—as state machines and as Simulink models—and illustrate how the notion of requirements coverage can be applied in these two instances.

4.3.1 Coverage of Finite State Machine Observers

To illustrate a synchronous observer, consider the state machine in Figure 8. This state machine encodes the requirement discussed in Section 4.1—our system behaves correctly as long at this state machines stays out of the error state (the state with the thick border to on the right). Such state machines can be constructed by hand to encode the required behavior or they can be automatically derived from property specifications. For example, there are algorithms automating the synthesis of state machines from temporal logic properties such as LTL discussed in the previous section.

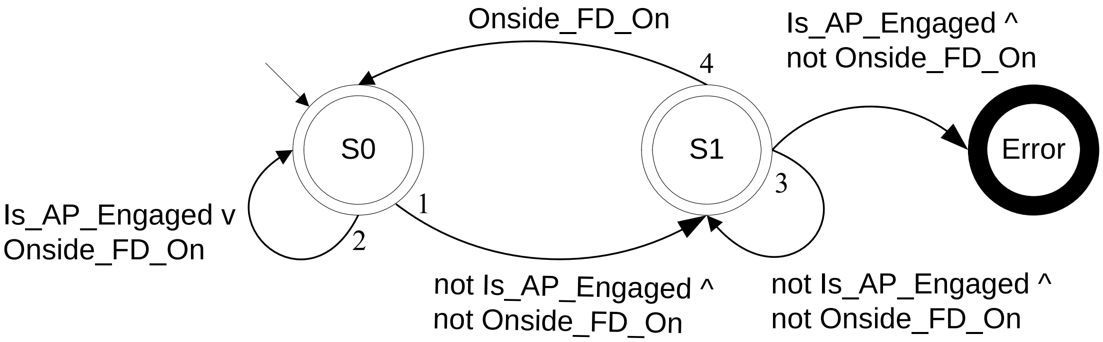

Figure 8: Property 1 captured as a simple finite sate machine.

In Figure 8, the states with a double border are accepting states; any cyclic path through the automaton with an accepting state as part of the cycle is considered successful. The state out to the right is the error state and any trace of events that would take us to the error state would violate our required behavior. With this description of the requirement we can start to objectively assess how well a set of test-cases covers the requirement.

If our state machine in Figure 8 had observed the manually developed test-case in Table 2, the test-case would start in state S0 with the Onside FD on and the AP engaged. When both the Onside Flight Director has been turned off and the AP disengaged, the state machine would take transition 1 to state S1. As we engage the AP, if the model is correct, the Onside FD will come on and the state machine would take transition 4 back to S0. Since this is an accepting state and part of a cycle, it is an acceptable requirements-based test. Note that this state machine defines both the inputs (AP engaged) as well as the expected output (Onside FD); exactly what we would expect from the high-level requirements.

Studying the state machine in Figure 8 we see that the test-case we developed manually really only covers part of the acceptable behavior of our system; there are many paths through this state machine that describe valid behaviors of the system. Coverage of these paths would be an objective measure of high-level requirements test coverage that we might want to consider. For example, we might want to make sure we cover all lassos [53] (a lasso consists of a cycle free path from an initial state to an a accepting state followed by a cycle free path back to the same accepting state; the complete path will look like a lasso). This would force us to consider several additional test-cases. For example:

1. Turn the Onside FD off and disengage the AP (S0 → S1)
2. Leave the AP disengaged for one step (S1 → S1)
3. Verify that the Onside FD does not come on

The test-case again turns the Onside FD off and disengages the AP. This will cause the state machine to take transition 2 from S0 to S1. Now we leave the AP disengaged for one execution cycle. Since the Onside FD shall not be turned on in this case, will take transition 3 from state S1 back to S1. Since this is an accepting state we have a lasso and a valid test sequence. This sequence exercises the scenario where the AP is never engaged and, consequently, the Onside FD does not need to be turned on. As should be clear from this discussion, we now have a systematic way of developing high-level requirements-based test-cases as well as an objective way of measuring how well we have exercised the various high-level requirements.

Note here that there are numerous ways of capturing a requirement as a finite state machine and there are numerous algorithms to translate a formula into a finite state machine. The structure of the manually created finite state machine or the algorithm used to generate the state machine will influence the selection of test-cases. The state machine in Figure 8 was generated from an LTL formula with the aid of the SAL toolset from SRI International and SAL helped us derive a very compact description with complex conditions on the transitions (conditions involving *or* and *not*). A different state machine created with the algorithm described by Gerth, Peled, Vardi, and Wolper [22] would yield nine states and 33 transitions. In this case, however, the conditions on the transitions would be simpler. Naturally, the test-cases required to yield lasso-coverage would be very different depending on which state machine we used as a basis for the test-case generation; a larger automaton with more lassos may give us a better suite of test-cases. Alternatively, one could require that every lasso in Figure 8 be exercised up to MC/DC coverage. This would require multiple test-cases per lasso and presumably provide a better test-suite than simply requiring lasso-coverage. These tradeoffs in terms of test-suite size and effectiveness of test-cases is completely unexplored and we hope to address this issue from both a theoretical and empirical perspective in the future.

4.3.2 Coverage of Observers in Captured in the Specification Language

It is also possible to express synchronous observers in the native notation of a model-based development or test generation tool. For instance, Reactis by Reactive Systems Inc. [48] uses Simulink [39] or Stateflow [40] models for observers, and the SCADE [17] tool from Esterel Technologies capture observers as additional SCADE nodes. For example, we can define Property 1 as an observer in the Simulink notation as shown in Figure 9.

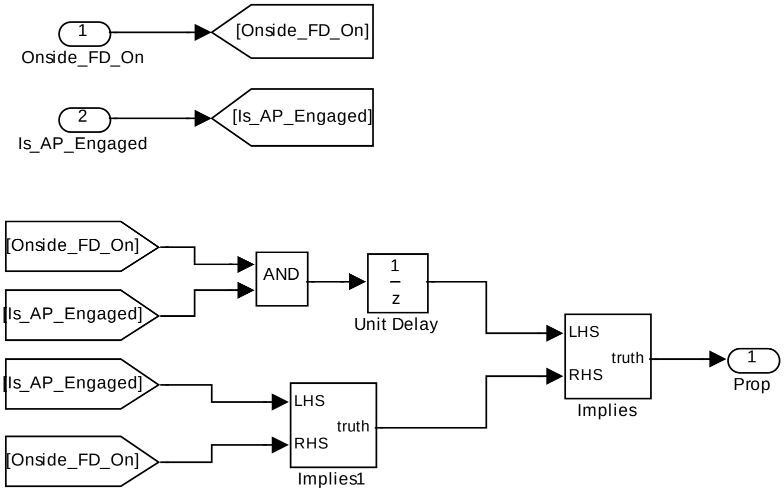

Figure 9: Our sample requirement captured as a Simulink model.

We can now measure coverage of this observer to determine how well this requirement has been exercised. As the property uses the same notation as the specification, the coverage metrics that have been defined for specifications, such as state, condition, and MC/DC, may be used to measure coverage of the requirement. However, it is an open question as to whether existing measures are adequate when models are used to define requirements.

4.4 Summary of Requirements-Based Tests

To summarize this section, viewing the high-level requirements as formal properties or observers expressed in some modeling language provides a mechanism where we can objectively measure how well our test-suite covers the requirements. To our knowledge, the notion of requirements coverage of software requirements has not been previously addressed in any formal way. As this is one of the goals called out in documents such as DO-178B [51], if suitable coverage measurements can be found, this capability would be a useful and important criterion for testing system behavior.

There are several areas related to requirements-based testing that merit further study. Since the notion of formal requirements-based testing is new, there is little experience with how to best capture the informal requirements as formal properties as well as what coverage criteria to use after the requirements have been formalized. Both issues naturally will have an impact on the test-cases developed and, consequently, on the effectiveness if the resultant test-suites. There is a great need for empirical studies to determine the effectiveness of test-suites constructed to various requirements coverage criteria.

5 Automating High-Level Requirements Test Generation

Given that the high-level requirements can be captured as formal properties and a model of the system behavior we can examine automating the process of deriving high-level requirements-based tests over the model.

Several research efforts have developed techniques for automatic generation of tests from the formal models. For example, the Critical Systems Research Group at the University of Minnesota have explored using model checkers to generate tests to provide structural coverage of formal models [30], [45], [46], [47]. Reactive Systems Inc. [48] has a commercial tool Reactis that uses heuristic search techniques to generate tests from Simulink [39] and Stateflow [40] models. Another commercial tool, T-VEC [9] uses a constraint-based approach to automatically generate test-cases from a number of different modeling languages, including Simulink.

In this section we will illustrate how requirements-based test-case generation can be automated using model-checking techniques (Section 5.1). Although this approach is currently in the prototyping stages, our initial experience shows great promise. To illustrate how requirements-based testing can be automated using commercial tools, we in Section 5.2 outline how the tool Reactis might be used for this task.

5.1 An Experiment in Requirements-Based Test Generation using Model Checkers

As mentioned above, in our research work we have focused on the potential of model-checkers as test-case generation tools. Model checkers build a finite state transition system and exhaustively explore the reachable state space searching for violations of the properties under investigation [14]. Should a property violation be detected, the model checker will produce a counterexample illustrating how this violation can take place. In short, a counterexample is a sequence of inputs that will take the finite state model from its initial state to a state where the violation occurs.

One way to use a model checker to find test-cases is by formulating a test criterion as a verification condition for the model checker. In the previous section, we described UFC over paths and defined sets of LTL formulas that were sufficient to show that a particular atomic condition affects the outcome of the property. Given this set, we can now challenge the model checker to find a way of generating a path to satisfying one of these formulas by asserting that there is no such path (i.e., negating the formula). We call such a formula a *trap formula* or *trap property* [21]. The model checker will now search for a counterexample demonstrating that this trap property is, in fact, satisfiable; such a counterexample constitutes a test-case that will show the requirements UFC obligation of interest over the model. By repeating this process for all formulas within the set derived from a property, we can derive UFC coverage of the property. By performing this process on all requirements properties of interest, we can derive a test-suite that generates UFC of requirements.

In previous work, tools were used to automatically generate tests to reach a particular state within a model or program. Here we are attempting to raise the level of abstraction to where tests are derived from high-level requirements. By providing a model of the system behavior and the requirements captured as properties, it is possible to derive test-cases that illustrate how the required properties hold in the model. This leads to automatically generated test-cases that can immediately be traced back to high-level requirements, a stated objective in DO-178B [18]. Since traceability to high-level requirements is so important when developing and testing safety-critical systems, these ideas discussed here have the potential to significantly improve requirements-derived testing while reducing cost.

In order to assess the feasibility of auto generating tests from requirements on a moderately-sized example, we generated a set of requirements-based tests from the FGS case example described in Section 2.1 as well as in Appendix A The test-cases from requirements were generated in two ways. First, we simply negated the formal requirements (LTL properties) and provided them as trap properties to the model checker; the resulting test-suite provides one test-case for each requirement. Second, we generated trap properties for the positive UFC obligations discussed in Section 4.2 and used the model checker to generate UFC-adequate tests over the requirements. In this initial experiment we were interested in determining (1) the feasibility of generating such tests with a model checker, (2) the number of test-cases needed to provide requirements UFC coverage for a substantial and realistic example, and (3) what coverage of the model these test-sets would provide.

To provide realistic results, we conducted the case study using the requirements and model of the close to production model of a flight guidance system we introduced earlier in the paper. This example consists of 293 informal requirements formalized as LTL properties as well as a formal model captured in our research notation RSML^{-e}.

5.1.1 Experimental Setup

The case study followed 3 major steps:

Trap Property Generation:

We started with the 293 requirements of the FGS expressed formally as LTL properties. We generated two sets of trap properties from these formal LTL properties.

First, we generated tests to provide *requirements coverage*; one test-case per requirement illustrating one way in which this requirement is met. We obtained these test-cases by simply negating each requirement captured as an LTL property and challenged the model checker to find a test-case.

Second, we generated tests to provide *requirements UFC coverage* over the syntax (or structure) of the required LTL properties. The rules for performing UFC over temporal operators were explained in Section 4.2. Using these rules, we implemented a transformation pass over the LTL specifications (we used the same implementation to generate tests for requirements coverage mentioned above). Presently, we only generate the positive UFC set over the temporal properties. The positive UFC set of properties is then negated to generate a set of trap properties. For this example we found it to be of no interest to investigate the negative cases since we knew all requirements were met in the model; therefore, no tests for the negative cases would have been found.

Test-suite Generation:

To generate the test-cases we leveraged a test-case generation environment we built in a previous project [30]. This environment uses the bounded model checker in NuSMV for test-case generation. We automatically translate the FGS model to the input language of NuSMV, generate all needed trap properties, and transform the NuSMV counterexamples to input scripts for NIMBUS—the execution environment for RSML^{-e}—so that the tests can be run on the model under investigation. Previously, we had enhanced the NIMBUS framework with a tool to measure different kinds of coverage over RSML^{-e} models [28]. Therefore, we could run the test-cases on the RSML^{-e} models and measure the resulting coverage.

Coverage Measurement:

We measured coverage over the FGS model in RSML^{-e}. We ran both test-suites (requirements coverage as well as requirements UFC coverage) over the model and recorded the model coverage obtained by each. We measured State coverage, Transition coverage, Decision coverage, and MC/DC coverage.

> **State Coverage:** (Often referred to as variable domain coverage.) Requires that the test set has test-cases that enable each control variable (Boolean or enumerated variable) defined in the model to take on all possible values in its domain at least once.
>
> **Transition Coverage:** Analogous to the notion of branch coverage in code and requires that the test set has test-cases that exercise every transition definition in the model at least once.
>
> **Decision Coverage:** Each decision occurring in the model evaluates to true at some point in some test-case and evaluates to false at some point in some other test-case.
>
> **Modified Condition and Decision Coverage (MCDC):** Every condition within the decision has taken on all possible outcomes at least once, and every condition has been shown to independently affect the decision's outcome.

5.1.2 Experimental Results and Analysis

We used our tools to automatically generate and run two test-suites for the FGS; one suite for requirements coverage and one for requirements UFC coverage. The results from our experiment are summarized in Table 11 and Table 12.

Criteria for generation → **Generation statistics ↓**	**Requirements Coverage**	**UFC Requirements Coverage**
Number of Trap Properties	293	887
Number of Test-cases	293	715
Time expended for test-suite generation	3 minutes	35 minutes

Table 11: Summary of the test-case generation results.

Table 11 shows the number of test-cases in each test-suite and the time it took to generate them. It is evident from the table that the UFC coverage test-suite is three times larger than the requirements coverage test-suite and can therefore be expected to provide better coverage of the model than the requirements coverage test-suite. Also, the time expended in generating the UFC coverage test-suite was significantly higher than the time necessary to generate the requirements coverage test-suite.

We observe that our algorithm generating UFC over the syntax of the requirements generated 887 trap properties. Nevertheless, only 715 of them generated counter examples[2]. For the remaining 172 properties, the UFC trap property is *valid,* which means that the condition that it is designed to test *does not* uniquely affect the outcome of the property.

There is an interesting parallel between our definition of unique effect of conditions and *vacuity detection* [6]. Intuitively, a model *M vacuously satisfies* property *f* if a subformula *g* of *f* is not necessary to prove whether or not *f* is true. Formally, a formula is vacuous if we can replace *g* by any arbitrary formula φ in *f* without changing the satisfaction of *f*:

$$M \models f \Rightarrow M \models f[g \leftarrow \varphi]$$

Our trap properties are essentially checks of vacuity for each condition subformula *g* within a formula *f*. If a trap property is falsifiable, then we have found a witness that shows that *g* cannot be replaced by an arbitrary φ. If a trap property is valid, then there is no path through the model in which the value of *g* ultimately affects the validity of the original formula *f*.

[2] We initially used the NuSMV bounded model checker with depth 5, that is, we only looked for test-cases with a length of 5 steps or shorter. If the bounded model checker did not find such a test-case, then we provided the trap property to the symbolic model checker in NuSMV and found that in all cases, it verified the property as being true; that is, there is no test-case for this particular MC/DC obligation.

If a formula contains vacuity, then it is likely that there is a problem with the model *M* or the property *f*. In our experience, we have found two main reasons why some formulas are vacuous:

1. The model is incorrect. That is, the scenario for which we cannot find a test-case is valid and a test-case should exist. This situation might occur because the system model is specified incorrectly or because the invariants and environmental assumptions made when developing the model are more restrictive than necessary and hence do not allow the behavior of interest.
2. The requirements are weaker than necessary—a case we found prevalent in this example. Here, the model under investigation may be correct and more restrictive than the behavior defined in the original requirement; the original requirement is simply incorrect and allows behaviors that should not be there. Consequently, any attempt to generate test-cases to cover these erroneous behaviors will fail.

In Table 12 we show the coverage of the formal model achieved when running the test-cases providing requirements coverage and requirements UFC coverage respectively. We measured four different model coverage criteria as mentioned in Section 5.1.1.

Criteria for generation → **Coverage of model ↓**	**Requirements Coverage**	**UFC Requirements Coverage**
State Coverage	37.19 %	99.12%
Transition coverage	31.97 %	99.42%
Decision coverage	46.42 %	83.02%
MC/DC coverage	0.32 %	53.53%

Table 12 :
Summary of the model coverage obtained by running the requirements based tests generated to requirements coverage as well as requirements MC/DC coverage.

The results in Table 12 show that the test-suite generated to provide requirements coverage (one test-case per requirement) gives very low state, transition, decision coverage, and almost no MC/DC coverage. This is in part due to the structure of the properties. Most of the requirements in the FGS system are of the form

$$G(a \rightarrow X\, b)$$

Informally, it is always the case (G) that if *a* holds in some state, in the next state (X), *b* will hold. The test-cases found for such properties are generally those in which the model goes from the initial state to a state where *a* is false, thus trivially satisfying the requirement. Such test-cases exercise a very small portion of the model and the resultant poor model coverage is not at all surprising.

On the other hand, the test-suite generated for UFC over the syntax of the specification provides very high state, transition, and decision coverage. Nevertheless, the test-suite generated low MC/DC coverage over the model. After some thought, it became clear that this is largely due to the structure of the requirements defined for the FGS. Consider the requirement

"When the FGS is in independent mode, it shall be active"

This was formalized as a property as follows:

```
G(m_Independent_Mode_Condition.result-> X(Is_This_Side_Active = 1))
```

Informally, it is always the case that if the condition for being in independent mode is true, in the next state the FGS will always be active. Note here that the condition determining if the FGS is to be in independent mode is abstracted to a macro (Boolean function) returning a result (the .result on the left hand side of the implication). Many requirements for the FGS were of that general structure.

```
G(Macro_name.result → X b)
```

Therefore, when we perform UFC over this property structure, we do not perform MC/DC over the—potentially very complex—condition making up the definition of the macro since this condition has been abstracted away in the property definition.

Consider another requirement from the FGS (slightly simplified).

```
G(When_Turn_Modes_On.result → X b)
```

Informally, it is always the case (G) that if When_Turn_Modes_On.result holds, in the next state (X), *b* will hold. The Macro When_Turn_Modes_On defines when the flight modes are to be turned on and displayed.

The Macro When_Turn_Modes_On is defined in RSML^{-e} as:

```
MACRO When_Turn_Modes_On():
   TABLE
      Onside_FD = On  : T * *;
      Offside_FD = On : * T *;
      Is_AP_Engaged   : * * T;
   END TABLE
END MACRO
```

Since the structure of When_Turn_Model_On is not captured in the required property, the test-cases generated to cover the property will not be required to exercise the structure of the macro and we will most likely only cover one of the MC/DC cases needed to adequately test the macro.

Note that this problem is not related to the method we have presented in this report; rather, the problem lies with the original definition of the properties. Properties should not be stated using internal variables, functions, or macros of the model under investigation; to avoid a level of circular reasoning (using concepts defined in the model to state properties of the model) the properties should be defined completely in terms of the input variables to the model. If, for some reason, a property must be stated using an internal variable (or function) then additional requirements (properties) are required to define the behavior of the internal variable in terms of inputs to the system. In this example, a collection of additional requirements defining the proper values of all macro definitions should be captured. These additional requirements would necessitate the generation of more test-cases to achieve requirements UFC coverage and we would presumably get significantly better coverage of the model.

Even if the requirements would have been specified correctly we do not anticipate 100% MC/DC coverage of the model given 100% UFC coverage of the requirements; the set of requirements is most likely inadequate and the missing details added in the modeling process. In this case we may have to define additional requirements to address inadequacies in our property set.

5.2 Requirements-Based Test Generation using Reactis

The test-case generation techniques based on model-checking discussed in the previous section are very promising but, at the moment, they are experimental and further research is needed before they will be incorporated in commercial tools. To evaluate how a commercially available test-case generation tool can be used to perform requirements-based testing we attempted to use the tool Reactis from Reactive Systems Inc. [48] to this effect. In this section we provide an overview of this effort and discuss our findings.

Reactis uses random and heuristic search to try to generate test-cases up to some level of structural coverage of a Simulink model. Reactis defines a notion of MC/DC coverage on the model and uses this as its most rigorous notion of structural coverage. It also supports *assertions*, which are synchronous observers written in Simulink or StateFlow that encode requirements on the system. The assertion checking is integrated into the structural test generation that Reactis performs to generate model-based tests (described in more detail in Section 6). In this process, Reactis attempts to structurally cover the assertion (as well as the rest of the model) to the MC/DC metric defined on Simulink models.

In its current form, Reactis has several drawbacks for performing requirements-based testing. One is that the coverage of the requirement is integrated into a model-based testing phase, that is, Reactis will attempt to find MC/DC coverage of both the model and the requirement at the same time. This means that tests are not particularly well-structured; it is difficult to determine exactly what is being tested in a particular test-case. Also, this integration precludes separating out the requirements-based tests to determine how well the model is covered. Finally, the search algorithms used by Reactis are random and incomplete, and on many classes of models may not find test-cases showing structural coverage of a requirement even if they exist. For this reason, Reactis is also unable to detect vacuity in synchronous observers – it cannot determine whether a path is unreachable.

On the other hand, Reactis is able to analyze models that involve large-domain integers and reals that currently are beyond the capability of model checking tools. Also, it would be straightforward to modify the algorithms in Reactis to only attempt to provide structural coverage of assertions, rather than of the entire model. This would allow the kinds of comparisons of requirements and model coverage that we provided for the model checking tools.

We have not yet tried to use Reactis on a large-scale experiment, largely because it would require rewriting all of our CTL/LTL properties as synchronous observers. Nevertheless, as a first step, we have examined the scalability of Reactis—both in terms of time and of coverage achieved—on several models using a handful of synchronous observers. The results are described in Table 13.

Model	Conditional Subsystems	Branches	MC/DC coverage	Time to load	Time to generate suite.
Cruise	7	38	90%	Negligible	40 seconds
ASW	0	60	84%	Negligible	23 seconds
Honeywell Voter	0	137	47%	Negligible	48 seconds
WBS-No Failure	0	462	81%	Negligible	1 min 22 sec
FCS5000	511	2074	DNF	5 seconds	16 hours 35 minutes

Table 13 : Performance results using Reactis for test-case generation.

The first three columns are the name of the model and the number of conditional subsystems and branches in each model to provide an idea of the size of the models. Also listed is the amount of MC/DC test coverage achieved on the model by creating 5 random tests of 100 steps followed by a targeted phase with 20000 steps (the default options in Reactis). Finally, we list the time to load the model, and the time to generate the above mentioned test-suite on a 3.5GHz Pentium 4 with 3GB RAM

The most surprising result was that generating a subset of the targeted tests for the FCS5000 takes almost 17 hours. From this result, it appears that the test generation facility in Reactis does not scale linearly with the size of the model. Also, Reactis did not achieve good coverage of the Honeywell Voter model, which contains a significant number of non-linear constraints over floating-point variables. Note, however, that we are just starting to use Reactis and more experimentation must be performed to determine whether these results can be generalized.

5.3 Summary of Requirements-Based Test Generation

There is great promise in automatic test-case generation; the use of tools to automate a currently a manual and labor intensive process. Automation holds the promise of reduced cost and increased test quality. Nevertheless, there are several challenges that must be met. As pointed out in Section 4, it is not at all clear which requirements coverage criteria are suitable—their effectiveness in revealing modeling faults is completely unknown. In addition, we must explore the relationship between requirements-based structural coverage and model-based structural coverage. Given a “good” set of requirements properties and a test-suite that provides a high level of structural coverage of the requirements, is it possible to achieve a high level of structural coverage of the formal model and of the generated code? That is, does structural coverage at the requirements level translate into structural coverage at the code level? If it is possible to achieve a high-level of code coverage from structurally complete requirements-based tests, then it might be possible to auto generate tests that are immediately traceable to the requirements and that meet many of the regulatory requirements in standards such as DO-178B.

6 Model-Based Tests — Conformance Testing

As mentioned in the introduction to this report, when testing of the model has been completed (the validation process is complete) and we have derived an implementation from the model—either through manual coding or through code generation—the testing to demonstrate that the source code implements the model commences; it is time for implementation conformance testing. Since the tests in conformance testing are derived from the model, this testing is often referred to as model-based testing. In conformance testing, we are not interested in what the formal model does; we are interested in determining if the implementation correctly implements the formal model.

The nature of model-based conformance testing leads to several significant differences as compared to requirements-based validation testing. First, we assume that the model is correct and it can, therefore, be used as an oracle in the testing process; we want to show that for arbitrary input vectors, the formal model generates the same result as the implementation. Second, the requirements on the test-cases themselves vary between requirements-based testing and model-based testing. In requirements-based the test-cases must represent clear and realistic operational scenarios and it is highly desirable that the tests that check related features are similar in order to make test validation easier. When we perform model-based conformance testing, on the other hand, we are much less concerned about the characteristics of the test-cases; we simply want enough tests to convince us that the implementation is correct with respect to its specification.

The characteristics of model-based conformance testing makes it ripe for automation and numerous tools capable of generating tests from large models have been developed, for example, [3],[30],[21],[43],[44],[46],[47],[49]. These tools are based on different techniques to generate the test-cases, for example, random or heuristic search, constraint solving, theorem proving, or model checking, and in general they are quite effective. In a project funded through NASA Ames Research Center, the University of Minnesota has investigated the feasibility of using model checkers to automatically generate test-cases from formal models [30],[45],[46],[47]. Model checkers are effective as test-case generation tools since they (1) scale reasonably well to large systems, (2) generate complete test sequenced guiding the system from an initial state to the construct of interest, and (3) generate both the test inputs as well as the expected outputs. The techniques developed in this project have been adopted in the Methods and Tools for Light Critical Systems Project and were evaluated on models developed at Rockwell Collins.

Although we have had great success with model checkers as test-case generation tools, in this report we are not going to focus our attention on the method used for test-case generation—instead we are going to focus on two critical issues that have surfaced in our investigations. First, the coverage criteria commonly used in the source-code domain, for example, statement coverage, branch coverage, etc., and coverage suggested for the model domain, for example, transition coverage, variable domain coverage, etc., may be reasonable when *measuring* test coverage but seem inadequate when used to *generate* tests. Second, some test-case generation tools generate one test-case per test target. Other tools generate fewer test-cases but those test-cases each cover several test targets at the time. Our observations indicate that these reduced test sets find fewer faults than tests sets with one test-case for each target.

In the following sections, we discuss the experiments that led us to these observations and present the need for future investigations.

6.1 Specification Coverage Criteria

To evaluate the fault finding capability of automatically generated test-suites, we conducted an experiment using an RSML^{-e} model of the mode-logic of the Flight Guidance System (FGS) developed as part of the Methods and Tools for Flight Critical Systems Project [41] (briefly described in Appendix A). We used our framework for specification-based test generation based on the NuSMV model checker to generate test sets meeting six specification test adequacy criteria. The generated tests were then executed on versions of the specification with seeded faults to determine how well each suite was able to reveal faults as compared to a set of random tests.

Our experiment indicated that some common specification test adequacy criteria we evaluated are woefully *inadequate* and are not likely to reveal faults. Stronger criteria, for example, UFC over the specification, fared better and revealed all seeded faults in our experiment. We identified two reasons for the inadequacy of some—seemingly reasonable—criteria: (1) the structure of the flight guidance model leads the model checker to find test-cases that technically provide the right coverage but do not exercise the logic of the model, and (2) the semantics of the specification language make some coverage criteria unsuitable. We believe theses findings generalize to other systems and other formalisms and that automated test-case generation must be approached with caution. There is a great need for a rigorous evaluation of effectiveness of these specification coverage criteria.

In the following sections, we describe how we conducted our experiment and discuss the test coverage criteria used for this study. We analyze the results obtained from our experiments, discuss their implications, and identify further research opportunities.

6.1.1 Experimental Setup

In our experiment, we were interested in answering one question:

> *How well do the test-cases generated to meet various structural coverage criteria reveal faults as compared to random tests generated and run with the same effort?*

To answer this question, we devised an experiment evaluating the fault-finding capability of six specification coverage criteria: variable domain, transition, decision (with or without a requirement that the decision is actually evaluated in the execution—a discussion will follow below), and MC/DC (also with or without evaluation requirements) coverage.

As mentioned above, to provide realistic results, we conducted the experiment using the RSML^{-e} model of the flight guidance system we developed early in the Methods and Tools for Flight Critical Systems project. To provide targets for our testing effort, we created a collection of faulty specifications. These faulty specifications are intended to mimic the source code that would have been generated or manually implemented from a formal specification. The reason for us to use fault seeded specifications rather than fault seeded source code is that we had much better tools available to measure coverage of the specifications as opposed to source code. We do not believe the validity of our experiments was compromised because of this choice—the structure of the specifications and the nature of the possible fault types is very similar to the structure and fault types we would see in any source code derived from the specifications.

We first reviewed the revision history of the FGS model to understand what types of faults were removed during the original verification process. We then implemented a random fault seeder to inject representative faults to create a suite of faulty specifications. The faults we seeded fell into the following four categories:

> **Variable Replacement:** A variable reference was replaced with a reference to another variable of the same type.
>
> **Condition Insertion:** A condition that was previously considered a "don't care" in one of the RSML^{-e} tables was changed to true (the condition is required to be true).
>
> **Condition Removal:** A condition that was previously required to be true or false in an RSML^{-e} table was changed to "don't care".
>
> **Condition Negation:** A condition that was previously required to be true in a an RSML^{-e} table was changed to false, or vice versa.

We used our fault seeder to generate 100 faulty specifications (25 for each fault class). This process yielded 72 specifications that were semantically different than the original. We then performed the experiment by conducting the following steps for several different structural coverage criteria:

1. We used the original specification to generate a test-suite to a coverage criterion of interest, for example, transition coverage, and measured the effort involved.
2. We ran the test-suite on the 72 faulty specifications and recorded the number of faults revealed as well as the time required run the test-suite.
3. We used the same effort (the sum of the time used to generate as well as run the structural test) to generate and run a randomly generated test-suite. We generated this-suite using a statistical testing tool also implemented as part of our research.
4. Given the results of the previous steps, we compared the relative fault finding capability of the randomly generated tests versus the structural tests.

6.1.2 Coverage Criteria

A *test-case* is a sequence of values for the input variables of an RSML^{-e} specification. This sequence of inputs will guide the RSML^{-e} specification from its initial state to the structural element, for example, a transition the test-case was designed to cover. A *test-suite* is simply a set of such test-cases.

For the case study we selected six specification coverage criteria.

> **State Coverage:** (Often referred to as variable domain coverage.) Requires that the test set has test-cases that enable each control variable defined in the model to take on all possible values in its domain at least once.
>
> Consider, for example, the control variable ROLL in the FGS specification example:
>
> STATE_VARIABLE ROLL : { Cleared, Selected, UNDEFINED };
>
> A test-suite would achieve variable domain coverage on ROLL, if for each of its three different possible values, there is a test-case in which ROLL takes that value. Note that a single test-case might actually achieve this coverage by assigning different values to ROLL at different points in the sequence. Nevertheless, to provide a comprehensive test-suite in this case study we generated one test-case for each state variable value. The perils of generating fewer test-cases where each test-case covers several test targets are discussed in Section 6.2.
>
> **Transition Coverage:** Analogous to the notion of branch coverage in code and requires that the test set has test-cases that exercise every transition definition at least once.
>
> **Decision Coverage:** Each decision occurring in the model evaluates to true at some point in some test-case and evaluates to false at some point in some other test-case. Note that if the decision is, for example, in a function, there is no requirement that the function is actually invoked—this criterion only requires that the decision would have evaluated to true/false if it was evaluated during the test-case. We chose to include this criterion since the published versions of decision coverage could be interpreted this way by a careless analyst. When instrumenting a model (or program) to measure test coverage the evaluation of the constructs of interest (in this case a decision) is implicit; when generating test-cases using model-checking techniques the fact that the construct of interest must be evaluated must be explicit.
>
> **Decision Coverage with Single Uses:** Analogous to decision coverage, but the decision must actually be evaluated. For example, for a decision in a function, the decision must evaluate to true/false while the function is invoked from some point in the model.

Modified Condition and Decision Coverage (MC/DC): Every condition within the decision has taken on all possible outcomes at least once, and every condition has been shown to independently affect the decision's outcome (this criterion is discussed extensively in Section 4.2 of this report). Note again that invocation of the decision is not required.

MC/DC with Single Uses: Analogous to modified condition and decision coverage, but the decision must actually be evaluated.

6.1.3 Results

We generated tests to achieve the coverage discussed in the previous section. The tests we generated were then executed on our fault seeded models and the fault finding capability compared to that of tests generated randomly using the same effort as expended on each structural test set. The results are presented in Table 14. To our disappointment, the structural tests performed consistently worse than our random tests. The reasons for this are discussed in the next section.

Coverage Criteria	Test Set Size	Faults Found
Random	100	66
Variable Domain	115	32
Transition	313	64
Decision	435	67
Decision Usage	478	69
MC/DC	537	70
MC/DC Usage	334	72

Table 14: Faults found with the six coverage criteria as well as through random testing.

6.1.4 Discussion of Coverage Criteria

Our initial reaction to these results was to question our implementation of the test-case generation—we simply suspected that we erroneously generated tests that did not provide the desired coverage. A close examination (and specification coverage measures) confirmed that the test-suites did provide the desired coverage—some test-suites simply did not reveal many faults. Further inspection revealed that the problems were related to (1) a model structure that "cheats" the coverage criteria, and (2) some coverage criteria that are inadequate with respect to the semantics of our specification language (as well as other common languages). Below we elaborate on these two issues.

The test-cases we generated were universally very short and—with the exception of MC/DC—quite poor at revealing faults. To explain this phenomenon, let us take a closer look at the flight guidance system used in the study.

The FGS is part of a larger Flight Control System (FCS) that, among many other components, contains two FGSs—a left FGS and a right FGS (one for the pilot and one for the co-pilot). In most situations, only one FGS is actually flying the aircraft (it is the active FGS). The other FGS (the inactive FGS) behaves as a hot spare, receiving all of its state information from the active FGS. In short, there is an interface between the FGSs through which they can control each other and, if one side is the active FGS, command the other side into arbitrary state configurations.

Most test-cases that the model checker found took advantage of this particular feature of the FGS model. The test-cases made the FGS under test the inactive FGS and simply used the interface to the other FGS interface to drive the model to the desired state (or take the desired transition, or make the desired condition true or false). For example, when the FGS is active there are some rather complex rules for when to enter the ROLL mode. When the FGS is inactive, on the other hand, all that is required to enter ROLL mode is to command it there with an input variable. Thus, it is possible for test-cases to achieve coverage by *commanding* the FGS to go to states and take transitions—they do not exercise the *actual mode logic* of the FGS. Naturally, such test-cases will not reveal any faults seeded in the mode logic of the FGS. Unfortunately, this is exactly the test-case that current test-case generation tools are likely to find—it is the simplest possible way of covering the test target. The better performance of MC/DC is natural since it requires the logic to be fully exercised—we have to investigate the logic processing commands form the other FGS as well as the logic making mode decisions based on pilot inputs.

To reduce this problem, we can simply prohibit messages on the FGS to FGS interface and in that way force the model checker to find test-cases that actually use the mode logic. For example, we can add an invariant to the model checker prohibiting the FGS from being inactive. This is technically quite an easy thing to do, but it requires intimate knowledge of the existence of the FGS-FGS interface and the knowledge that this interface can be "abused" by the test-case generation automation to achieve domain, transition, and decision coverage. We see this as an issue in many critical systems since they are typically designed with one or more hot-spares that need to be kept synchronized.

To investigate the how severe this effect was on the fault finding capabilities of our test-suites; we regenerated the two worst performing test-suites—variable domain and transition coverage—suites with the above-mentioned invariant that keeps the FGS model in the active state. As can be seen in Table 15, the fault finding capability of the test-suites went up. The performance is still inadequate, however, and random testing performs better. From this work we have come to the conclusion that variable domain and transition coverage are clearly inadequate in this domain—more elaborate coverage is necessary.

	Var. Domain	**Var. Domain (a)**	**Transition**	**Transition (a)**
Faults Found	32	46	58	59

Table 15: Faults found when the FGS is forced to be in Active mode. The results when the FGS is forced active are indicated with the (a).

To illustrate the possible problems with test-case generation without precisely defined coverage criteria; let us discuss the problems with naïve interpretation of condition based coverage criteria. Consider a small code fragment:

```
1: A := b or c
2:
3: if (A and d) then <stmnts>
4: else <stmnts>
```

Two test inputs *(b,* ***not*** *c,* ***not*** *d)* and *(****not*** *b,* ***not*** *c,* ***not*** *d)* would achieve decision coverage of the decision on statement 1, but there is no requirement that the outcome of the decision on statement 1 affects the flow of the program on statement 3; in the definitions of decision coverage and MC/DC we have seen in the literature there have been no consensus on how the various decision coverage criteria shall be interpreted [27] [18].. A more formal treatment of how the group at the University of Minnesota interprets these coverage criteria can be found in [44] [45].

Consider a minor modification of the example above where we break out the decision on line 1 to a Boolean function.

```
1: func A():
2:      {return (b or c)}
3:
4: if (A() and d) then <stmnts>
5: else <stmnts>
```

In this example, we still achieve decision coverage of the decision on line 2 (and the outcome of the decision is still irrelevant). With a minor modification rearranging the decision on line 4 we are not changing the behavior of the program.

```
1: func A():
2:      {return (b or c)}
3:
4: if (d and A()) then <stmnts>
5: else <stmnts>
```

In this case, if the language has lazy-evaluation, A() will not get called and the decision (now on line 2) will not even get evaluated. Technically, however, with a naïve interpretation of decision coverage the two test inputs *(b,* ***not*** *c,* ***not*** *d)* and *(****not*** *b,* ***not*** *c,* ***not*** *d)* could be viewed as still achieving decision coverage of the decision on line 2. Naturally, these tests will not reveal any faults in the function since it will never get executed. Initially, we formalized the notion of decision coverage and MC/DC coverage in an overly simplistic way; we simply stated that each decision must evaluate in the desired way, but we did not make it explicit that the decision actually must be invoked during the program execution. As can be seen in Table 14, not requiring the invocation of the decision not surprisingly decreased the fault-finding capability of decision and MC/DC coverage. If the decision of interest is masked out or never evaluated, the test is not particularly useful. To our knowledge, the condition based coverage criteria required in practice (for example, in certification to DO-178B) require the evaluation of the decisions but do not require the outcome of a decision to actually influence the program or specification flow.

To solve this problem, the coverage criteria must be modified to take data flow into account—the criteria must assure that somehow the decisions are invoked *and* affect the outcome of the program execution. The unanswered question is "How?". We must modify the requirements on the test-suites to require that the decision be exercised in some way by the test-suite. For example, the following are possibilities:

1. Decision invoked at least once for each required truth assignment (the approach taken in DO-178B),
2. Every invocation of the decision is tested for each required truth assignment,
3. Every invocation is invoked for each truth assignment of the decision, and the invocation point must be demonstrated to have an independent impact on the control flow.

As can be seen above, there are many combinations of condition based and data-flow based coverage criteria that can be developed. The question is, which ones are likely to reveal faults and which ones will give us test-suites that are of a size that makes the test effort tractable? Our experiment showed that random testing performs better or equally well as some commonly used model coverage criteria—there is a clear problem with the structural specification coverage criteria that must be addressed through both theoretical and empirical studies.

6.2 Reduced Test Sets

As mentioned earlier, with model-based development comes the capability to generate large numbers of tests to use as conformance tests to provide assurance that the generated code is correct with respect to the specification from which it was generated. The cost of generating, executing, storing, and maintaining these test-suites can be reduced through *test-suite reduction techniques*. Test-suite reduction aims to remove (or not generate at all) test-cases from a test-suite in such a way that "redundant" test-cases are eliminated. For example, a reduced test-suite TR may provide the same structural coverage as a test-suite T with significantly fewer test-cases.

To investigate the effect of test-suite reduction in the domain of automatically generated conformance test-suites, we conducted an experiment where we compared the test-suite size and fault-finding capability of reduced test-suites generated to six different specification test-adequacy criteria. We used the FGS model with roughly the same experimental setup as in the previous experiment (discussed in Section 6.1).

Our results show that one can dramatically reduce our automatically generated conformance test-suites while maintaining desired coverage. We also found that the fault finding of these reduced test-suites was adversely affected, and that the reduction is quite significant in the domain of model-based testing.

Our results point to two issues that merit further study. On one hand, the results indicate that test-suite reduction of test-suites providing structural coverage may not be an effective means of reducing testing effort—the cost in terms of loss in fault finding capability is too high. Second, and more importantly, the results cast additional doubts on the effectiveness of structural coverage criteria in general; test-suites that achieve structural adequacy criteria minimally do not seem effective and we must either discover ways to somehow augment them to enhance their effectiveness or define structural coverage criteria that are not as sensitive to test-suite size.

6.2.1 Experimental Setup

To investigate the relationship between test reduction and fault finding capability in the domain of model-based tests, we designed this experiment to test two hypotheses:

> ***Hypothesis 1:*** *Test reduction of a naively generated specification based test-set can produce significant reduction in terms of test-set size.*
>
> ***Hypothesis 2:*** *Test reduction will adversely affect the fault finding capability of the resulting test set.*

We formulated our hypotheses based on two informal observations. First, in a previous study we got an indication that one could achieve equivalent transition and state coverage with approximately 10% of the full test-set generated [30] and we believe this generalizes to other criteria as well. Second, intuitively, more tests-cases ought to reveal more faults. Only an extraordinarily good test adequacy criterion would provide a fault-finding capability that is immune to variations in test-suite size, and we speculate that none of the known coverage criteria possess this property.

In our experiment, the aim was to determine how well a test-suite generated to provide a certain structural or condition based coverage reveals faults as compared to a reduced test-suite providing the same coverage. To provide realistic results, we yet again used our trusted FGS model.

We conducted the experiment through the steps outlined below:

1. We used the original FGS specification to generate test-suites to various coverage criteria of interest, for example, transition coverage or MC/DC. Note here that we did this naively in that we generated a test-case for each construct we needed to cover. Thus, the test-suites were straightforward to generate, but they were also highly redundant.
2. We used the same 72 faulty specifications from the previous experiment.
3. We ran the full (non-reduced) test-suite on the 72 faulty specifications and recorded the number of faults revealed.
4. We generated and ran five reduced test-suites for each full test-suite, ensuring that the desired coverage criterion was maintained. In this experiment we used the same six specification coverage criteria

discussed in Section 6.1.1. As discussed below, we generated five reduced sets for each full test-suite to avoid skewing our results because we were lucky (or unlucky) in the selection of tests for a reduced test-suite.

5. Given the results of the previous steps, we compared the relative fault finding capability of the full test-suites versus the reduced test-suites.

To find reduced test-suites we used a heuristic algorithm. Finding a minimal test-suite that satisfies the test requirements is in general a very difficult problem (an NP problem [20]), but often greedy heuristics suffice to generate significantly reduced test-suites. The method we use begins with an empty set of test-cases and initializes the coverage to zero. The greedy algorithm then randomly picks a test-case from the full test-suite, runs the test, and determines if the test-case improved the overall coverage (for whatever criterion we are interested in). Any test-case that improves the coverage is added to the reduced set. This continues until we have exhausted all the test-cases in the full test-suite—we now have a, hopefully, much smaller suite that has the same coverage as the full test-suite.

Note that we randomly select test-cases from the full set to create a reduced test-suite. We then generate five separate reduced test-suites for each full test-suite. We choose this approach to reduce problems related to skewing the results by accidentally picking a “very good"” (or bad) set of test-cases. The results for all test runs are included in this report.

6.2.2 Results

As a baseline for our experiment, we ran the full test-suites as well as a randomly generated test-set. The results were presented in the previous section but for ease of reading this report they are reproduced in Table 16. The table shows the number of test-cases in each test-suite and their fault finding capability.

Coverage Criteria	Test Set Size	Faults Found
Random	100	66
Variable Domain	115	32
Transition	313	64
Decision	435	67
Decision Usage	478	69
MC/DC	537	70
MC/DC Usage	334	72

Table 16: Number of test-cases and number of faults found with test-suites to our six coverage criteria as well as a randomly generated test-suite.

As mentioned earlier, we generated five different reduced test-suites to control the possibility that we by chance got a very “good” or very “poor” reduced test-suite. The results of the reduction algorithm can be seen in Table 17.

Coverage Criteria	Full Set Size	Reduced Size (avg.)	Reduction
Variable Domain	115	20.2	82%
Transition	313	37.6	88%
Decision	435	44.0	90%
Decision Usage	478	41.6	91%
MC/DC	537	32.4	94%
MC/DC Usage	334	31.6	91%

Table 17: Average test-suite size after "greedy" reduction.

The results support our first hypothesis that test reduction results in significant savings in terms of test-suite size. In all cases, there was at least an 80% average reduction in the size of the test-suite. This reduction reinforces the findings in [57],[58],[50],[36] and is to be expected since our test-case generation method produces a significant number of overlapping test-cases (we generate a separate test-case for each construct of interest). Of more interest is the fault finding ability of the reduced tests-suites.

The fault finding capability of the full as well as reduced test-suites is summarized in Table 18. The results are in agreement with our second hypothesis that test-suite reduction will adversely impact the fault-finding ability of test-suites that are derived from synchronous data-flow models.

Coverage Criteria	Faults Detected		
	Full Set	Reduced Set (avg)	Reduction
Variable Domain	32	27.0	15.6%
Transition	64	58.0	9.38%
Decision	67	61.6	8.06%
Decision Usage	69	62.6	9.28%
MC/DC	70	63.2	9.71%
MC/DC Usage	72	66.8	7.22%

Table 18: Fault finding capability of the reduced test-suites.

As shown in Table 18, the number of faults detected by the reduced test-suites is significantly less for all coverage criteria that were examined in our experiment. In all cases there was at least a 7% reduction in the fault detection effectiveness. One may argue that a 7% reduction is rather small, but for our domain of interest, automated code generation in critical systems, any reduction in fault-finding ability is undesirable.

From our results we can also observe that the most rigorous coverage criteria, MC/DC with Usage, seems to be the least sensitive to the effect of test-suite reduction. We speculate that this is because it is simply harder to come up with a test-suite that provides this high level of coverage without finding faults—MC/DC with Usage is simply a "better" coverage criterion than the other ones we used in our experiment. We hypothesize that MC/DC with Usage is better than the other criteria in two respects. First, it seems to find more faults than any other criteria. Second, it seems to be less sensitive to the effect of test-suite reduction.

6.2.3 Discussion on Test-Suite Reduction

To summarize the findings, reduction of test-suite size has an unacceptable effect on the suite's fault-finding capability. Should there be an urgent need to reduce the test-suite size because of resource limitations we speculate that *test-case prioritization* [36] would be a better approach than test-suite reduction. In test-case prioritization, we would not eliminate any test-cases from our test-suite; we would instead attempt to *sort* the test-cases based on expected fault finding potential and execute the ones deemed to be most likely to reveal faults first. We would terminate the testing when our resources are depleted. Naturally, more work is needed to determine how to prioritize test-cases and also empirically evaluate if the test-case prioritization approach in fact performs better than reduced or minimized test-suites.

6.3 Summary of Model-Based Testing Experiments

To evaluate the test-suites generated to structural specification coverage criteria we conducted an experiment where we compared the fault-finding capability of these test-suites with randomly generated test-data. To provide a fair comparison, we assured that the effort spent on structural tests was comparable to the effort spend performing random tests. To our disappointment, we found that the structural tests often performed worse than randomly generated tests. The poor performance of the structural tests is, in this case-study, related to two issues (1) the structure of the flight guidance system (FGS) under test and (2) a mismatch between the formalization of the specification coverage criteria and the semantics of the specification language.

Part of the FGS functionality allows it to be *commanded* into arbitrary states and to take arbitrary transitions. This functionality is required when the FGS operates as a redundant spare to the second FGS on the flight deck. By using this interface, it is quite easy to satisfy the state and transition coverage criteria without actually exercising any of the "real" logic in the system under test and we are unlikely to reveal many faults in this logic. Condition based coverage criteria, such as the decision criterion we used in this experiment, as described in the literature, do not require that the decision of interest actually has an outcome on the control flow of the system under test. Therefore, it is easy to derive tests that will not reveal any faults in decisions that are masked out.

In a second experiment we investigated the effect of test-suite reduction in the domain of automatically generated conformance test-suites. Again we used the Flight Guidance System seeded with "representative" faults. Our results confirm our two hypotheses; one can dramatically reduce the automatically generated conformance test-suites while maintaining desired coverage, and the fault finding of the reduced test-suites was adversely affected. Our results cast doubts on the effectiveness of structural coverage criteria in general. Small (or minimal) test-suites that provide structural coverage criteria do not seem to be effective; we must understand better the reasons behind this problem and either discover ways to somehow augment the test-suites to enhance their effectiveness or define structural coverage criteria that are not sensitive to test-suite size. Our results indicate that more rigorous criteria, such as MC/DC, provide a better fault finding capability both for the full test-suites as well as the reduced test-suites as compared to less rigorous criteria, such as variable domain and transition coverage. This hints that MC/DC is more robust to the detrimental effect of test-suite size, but further investigations into the properties of structural coverage criteria are clearly needed.

Our experiences from this experiment raise some concern about the use of automated test-case generation from formal specifications. Effective test-case generation clearly requires an intimate knowledge of the structure and behavior of the system under test to ensure that the test-case generator actually creates high quality tests. The coverage criteria used in specification testing and specification based testing must also be refined to better fit the semantics of the specification languages and the structure of the models captured in these languages. In particular, there is a need to include some notion of data-flow information in the condition based coverage criteria. Finally, based on our results, we are skeptical of test-suite reduction techniques that aim solely to maintain structural coverage, because, in our opinion, there is an unacceptable loss in terms of test-suite quality. Thus, we advocate research into test-case prioritization techniques and experimental studies to determine if such techniques can more reliably lessen the burden of the testing effort by running a subset of an ordered test-suite as opposed to a reduced test-suite, without loss in fault-finding capability.

7 Conclusions and the Future Challenges

The move from traditional to model-based development of software provides opportunities as well as challenges for validating and verifying critical software systems. In this report, we point out that model-based development is a natural evolution of the traditional development process. In our view, the informal high-level software requirements are derived from the system requirements, safety requirements, and system architecture. The model is developed based on these requirements and captures the behavior described in the high-level requirements. Any additional information needed for implementation not captured in the high-level requirements is captured in the model; the model replaces the low-level requirements. The high-level requirements are formalized as properties that can then be verified to hold in the model. We view this collection of artifacts to be the requirements specification of the system under development.

The model-based development paradigm offers several new opportunities for testing, including the opportunity to provide objective measures of requirements coverage (Sections 4.2 and 4.3) and model coverage (Section 6.1) in addition to the traditional code coverage measured today. Furthermore, the formal models and properties enable us to automate large portions of the test-case development and test execution process (Sections 5 and 6.1.1).

Naturally, there are research challenges that must be overcome before we can reap the full benefits of model-based development and the automated testing approaches outlined above. First, the notion of requirements-based tests as discussed in Section 4 has not previously been explored and both effectiveness in terms of fault finding and availability of tool support needs to be explored further. Second, the coverage criteria that might be used in model-based testing are not well understood. Our investigations have raised serious concerns about the fault-finding ability of test-suites constructed to provide common coverage criteria as well as the common technique of test-set reduction (Section 6).

7.1 Requirements-Based Testing

As mentioned in Section 4, the model-based development paradigm offers the opportunity to provide objective measures of requirements coverage (Sections 4.2 and 4.3) in addition to the traditional code coverage and model coverage criteria used today. To the best of our knowledge, the notion of objectively measuring the test coverage of high-level software requirements has not been previously addressed in a systematic manner. Furthermore, the formal models and properties enable us to automate large portions of the test-case development and test execution process (Section 5).

The project discussed in this report only provides the start of the rigorous exploration of requirements-based testing; we have merely defined the notion and explored the feasibility of the approach. There are several topics that require further study.

1. Requirements formalization. Since formalizing high-level requirements is a rather new concept not generally practiced, there is little experience with how to best

capture the informal requirements as formal properties. What formalism and notation will be acceptable to the practicing developers, requirements engineers, and domain experts? In our work we have used CTL and LTL, but we are convinced that there are notations better suited to the task at hand.

2. Requirements Coverage Criteria. Since there—to our knowledge—has been no other work on defining coverage criteria for high-level software requirements, we do not know what coverage criteria will be useful in practice. We must find coverage criteria that (1) help us derive effective tests in threat the tests ate likely to reveal problems in the model derived from the high-level requirements and (2) provide test-suites that are of a reasonable size. There is a great need for empirical studies to determine the effectiveness of various test-case generation techniques on realistic examples.

3. Requirements versus Model Coverage. We must explore the relationship between requirements-based structural coverage and model-based structural coverage. Given a ``good" set of requirements properties and a test-suite that provides a high level of structural coverage of the requirements, is it possible to achieve a high level of structural coverage of the formal model and of the generated code? That is, does structural coverage at the requirements level translate into structural coverage at the code level? If it is possible to achieve a high-level of code coverage from structurally complete requirements tests, then it might be possible to auto generate tests that are immediately traceable to the requirements and that meet many of the regulatory requirements. Again, there is a need for empirical studies on production systems.

7.2 Specification-Based Testing

Modern tools allow us to generate vast numbers of test-cases from formal models. Although this automation has enormous promise, there are several longer term research challenges falling outside this project that must be addressed before we can reap the full benefits of model-based testing.

1. There is a great need to rigorously evaluate existing model coverage criteria and develop new ones suitable in this domain. Which criteria are likely to reveal model faults? Which criteria are likely to reveal translation or implementation faults? To our knowledge, with the exception of our own investigations reported here, there has been no comprehensive work in this area.

2. The relationship between coverage criteria and model structure is not well understood and merits further study. For example, MC/DC is more effective on complex conditions than if the same condition has been restructured using temporary variables to store intermediate results. This relationship between model structure, coverage criteria, and faultfinding must be thoroughly explored.

3. The procedure used to generate test-cases seems to have an impact of the quality of the tests. For example, our use of model checking techniques seems effective in terms of finding test-cases, but the test-cases are generally very short and may

have poor fault finding capability. A better understanding of the characteristic of desirable test-cases, for example, test-case length, is needed.

4. With the formal model we have the ability to generate both test inputs as well as the expected output (oracle data) for conformance testing. To make full use of this information we also need an *oracle procedure*, that is, a mechanism of comparing the expected output with the actual output from the system. There are several challenges with respect to this oracle procedure. For example, do we compare only the expected outputs, or do we also compare internal state changes? Slight, but acceptable, differences in the expected and actual outputs may appear due to differences in the representation of integers and real numbers in the implementation and the model; how do we accommodate such differences? Slight, but again acceptable, timing mismatches between what is predicted by the model and what appears in the implementation may be present; how are such differences accommodated? Not adequately addressing these issues may lead to potentially overwhelming numbers of reports of failed tests when the outcome of the test in fact was acceptable.

As outlined above, there are many research challenges that need to be addresses to reap the full benefits of model-based development, model-based testing, and test automation. In future projects we hope to get the opportunity to better explore these very promising—but inadequately understood—testing methods.

8 References

[1] ARP 4754. Certification Considerations for Highly-Integrated or Complex Aircraft Systems. SAE International, November 1996.

[2] P. Ammann and P. Black. A Specification-based Coverage Metric to Evaluate Test Sets. Proceedings of the Fourth IEEE International Symposium on High-Assurance Systems Engineering, November 1999.

[3] P Ammann, P. Black, and W. Majurski. Using Model Checking to Generate Tests from Specifications. Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98), pages 46-54, November 1998.

[4] M. Archer, C. Heitmeyer, and S. Sims. TAME: A PVS Interface to Simplify Proofs for Automata Models. User Interfaces for Theorem Provers, 1998.

[5] R. Armoni, D. Bustan, O. Kupferman, and M. Y. Vardi. Aborts vs. Resets in Linear Temporal Logic. TACAS 2003, pages 65-80, April 2003.

[6] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient Detection of Vacuity in ACTL Formulas. Formal Methods in System Design, 18(2): pages 141-162, 2001.

[7] S. Bensalem, P. Caspi, C. Parent-Vigouroux, and C. Dumas. A Methodology for Proving Control Systems with Lustre and PVS. Proceedings of the Seventh Working Conference on Dependable Computing for Critical Applications (DCCA 7), pages 89-107, January 1999.

[8] V. Berzins, Luqi, and A. Yehudai. Using Transformations in Specification-Based Prototyping. IEEE Transactions on Software Engineering, 19(5): 436-452, May 1993.

[9] M. R. Blackburn, R. D. Busser, and J. S. Fontaine. Automatic Generation of Test Vectors for SCR-style Specifications. Proceedings of the 12th Annual Conference on Computer Assurance (COMPASS'97), June 1997.

[10] J. Callahan, F. Schneider, and S. Easterbrook. Specification-Based Testing using Model Checking. Proceedings of the SPIN Workshop, August 1996.

[11] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese. Model Checking Large Software Specifications. IEEE Transactions on Software Engineering, 24(7): 498-520, July, 1998.

[12] J. J. Chilenski and S. P. Miller. Applicability of Modified Condition/Decision Coverage to Software Testing. Software Engineering Journal, pages 193-200, September 1994.

[13] Y. Choi and M. P. E. Heimdahl. Model Checking RSML^{-e} Requirements. Proceedings of the 7th IEE/IEICE International Symposium on High Assurance Systems Engineering, pages 109-118, October 2002.

[14] E. M. Clarke, O. Grumberg, and D. Peled. Model Checking. MIT Press, 1999.

[15] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, D. Van Campenhout, Reasoning with Temporal Logic on Truncated Paths, Proceedings of Computer Aided Verification (CAV) 2003, pages 27-39, 2003.

[16] A. Engels, L. M. G. Feijs, and S. Mauw. Test Generation for Intelligent Networks using Model Checking. Proceedings of TACAS'97, LNCS 1217, pages 384-398. Springer-Verlag, 1997.

[17] Esterel Technologies. SCADE Suite Product Description, http://www.esterel-technologies.com/v2/scadeSuiteForSafetyCriticalSoftwareDevelopment/index.html

[18] FAA Certification Authorities Software Team. What is a "Decision" in Application of Modified Condition/Decision Coverage and Decision Coverage (DC)?. Technical Report position paper, 2002.

[19] J. Fitzgerald and P. Gorm Larsen. Modeling Systems: Practical Tools and Techniques in Software Development. Cambridge University Press, 1998.

[20] M. R. Garey and M. R. Johnson. Computers and Intractability. Freeman, New York, 1979.

[21] A. Gargantini and C. Heitmeyer. Using Model Checking to Generate Tests from Requirements Specifications. ACM SIGSOFT Software Engineering Notes, 24(6): 146-162, November 1999.

[22] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple On-the-Fly Automatic Verification of Linear Temporal Logic. Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV, pages 3-18, 1996.

[23] O. Grumberg and D. E. Long. Model Checking and Modular Verification. ACM Transactions on Programming Languages and Systems, 16(3):843-871, May, 1994.

[24] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, The Synchronous Dataflow Programming Language LUSTRE, Proceedings of the IEEE, 79(9): 1305-1320, September 1991.

[25] G. Hamon, L. de Moura, and J. Rushby. Generating Efficient Test Sets with a Model Checker. Second International Conference on Software Engineering and Formal Methods, pages 261-270, September 2004.

[26] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. IEEE Transactions on Software Engineering, 16(4): 403-414, April 1990.

[27] K.J. Hayhurst and D.S. Veerhusen and L.K. Rierson. A Practical Tutorial on Modified Condition/Decision Coverage. NASA/TM-2001-210876, 2001.

[28] M. P.E. Heimdahl and D. George. Test-suite reduction for model based tests: effects on test quality and implications for testing Heimdahl. Proceedings. 19th International Conference on Automated Software Engineering, Pages 176 – 185, 2004.

[29] M. P.E. Heimdahl and N. G. Leveson. Completeness and Consistency in Hierarchical State-Based Requirements. IEEE Transactions on Software Engineering, 22(6): 363-377, June 1996.

[30] M. P. E. Heimdahl, S. Rayadurgam, W. Visser, G. Devaraj, and J. Gao. Auto-Generating Test Sequences using Model Checkers: A Case Study. Third International Workshop on Formal Approaches to Testing of Software (FATES 2003), 2003.

[31] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A Toolset for Specifying and Analyzing Requirements. Proceedings of the Tenth Annual Conference on Computer Assurance (COMPASS'95), 1995.

[32] C. Heitmeyer, R. D. Jeffords, and B. Labaw. Automated Consistency Checking of Requirements Specifications. ACM Transactions on Software Engineering and Methodology, 5(3):231-261, July, 1996.

[33] C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications. IEEE Transactions on Software Engineering, 24(11):927-948, November 1998.

[34] H. Hong, I. Lee, and O. Sokolsky and H. Ural A Temporal Logic Based Theory of Test Coverage and Generation. Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '02)}, April 2002

[35] R. Jasper, M. Brennan, K. Williamson, B. Currier, and D. Zimmerman. Test Data Generation and Feasible Path Analysis. Proceedings of the International Symposium on Software Testing and Analysis, pages 95-107, August 1994.

[36] J. A. Jones and M. Harrold. Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage. IEEE Transactions on Software Engineering, pages 195-209, March 2003.

[37] E. A. Lee. Overview of the Ptolemy Project. Technical Memorandum UCB/ERL M03/25, University of California, Berkeley, CA, 94720, USA, July 2003.

[38] Z. Manna and A. Pnueli. Temporal Verification of Reactive Systems: Safety. Springer-Verlag, New York, 1995.

[39] The Mathworks, Inc. Simulink Product Description, http://www.mathworks.com/products/simulink/

[40] The Mathworks, Inc. StateFlow Product Description, http://www.mathworks.com/products/stateflow/

[41] S. Miller, A. Tribble, T. Carlson and E. J. Danielson, Flight Guidance System Requirements Specification, NASA/CR-2003-212426, June 2003. Available at http://techreports.larc.nasa.gov/ ltrs/refer/2003/cr/NASA-2003-cr212426.refer.html.

[42] S. Miller, A. Tribble, M. Whalen, and M. P. E. Heimdahl, Proving the Shalls: Early Validation of Requirements Through Formal Methids, Journal of Software Tools for Technology Transfer, [in press].

[43] A. J. Offutt, Y. Xiong, and S. Liu. Criteria for Generating Specification-Based Tests. Proceedings of the Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'99), October 1999.

[44] S. Rayadurgam. Automatic Test-case Generation from Formal Models of Software. Ph.D. Dissertation, University of Minnesota, November 2003.

[45] S. Rayadurgam and M. P. E. Heimdahl. Coverage Based Test-Case Generation Using Model Checkers. Proceedings of the 8th Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS 2001), pages 83-91, April 2001.

[46] S. Rayadurgam and M. P. E. Heimdahl. Test-Sequence Generation from Formal Requirements Models. Proceedings of the Sixth IEEE International Symposium on High-Assurance Systems Engineering (HASE'2001), October 2001.

[47] S. Rayadurgam and M. P. E. Heimdahl. Generating MC/DC Adequate Test Sequences Through Model Checking. Proceedings of the 28th Annual IEEE/NASA Software Engineering Workshop (SEW'03), December 2003.

[48] Reactive Systems, Inc. Reactis Product Description. http:// www.reactive-systems.com/index.msp.

[49] D. J. Richardson, O. O'Malley, and C. Tittle. Specification-based Test Oracles for Reactive Systems. Proceedings of the Third International Symposium on Software Testing, Analysis, and Verification, pages 86-96, ACM Press, December 1989.

[50] G. Rothermel and M.J. Harrold and J. Ostrin and C. Hong. An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test-suites. Proceedings of the International Conference on Software Maintenance, November 1998.

[51] RTCA. Software Considerations in Airborne Systems and Equipment Certification (DO-178B). RTCA, 1992.

[52] J. M. Spivey. The Z Notation: A Reference Manual. Prentice Hall, 1992.

[53] L. Tan, O. Sokolsky, and I. Lee, Specification-based Testing with Linear Temporal Logic, IEEE Int. Conf. on Information Reuse and Integration (IEEE IRI-2004), Nov 8-10 2004.

[54] J. M. Thompson and M. P.E. Heimdahl. An Integrated Development Environment Prototyping Safety Critical Systems. Tenth IEEE International Workshop on Rapid System Prototyping (RSP) 99, pages 172-177, June 1999.

[55] J. M. Thompson, M. P.E. Heimdahl, and S. Miller Specification Based Prototyping for Embedded Systems. Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering, pages 163-179, September 1999.

[56] T-VEC Technologies. T-VEC Product Description, http://www.t-vec.com/solutions/products.php

[57] W.E. Wong and J.R. Horgan and A.P. Mathur and A.Pasquini. Test Set Size Minimization and Fault Detection Effectiveness: A Case Study in a Space Application. Proceedings of the 21st Annual International Computer Software and Applications Conference, August 1997

[58] W.E. Wong and J.R. Horgan and S.London and A.P. Mathur. Effect of Test Set Minimization on Fault Detection Effectiveness. Software Practice and Experience, pages 347-369, April 1998.

Appendix A - The Flight Guidance System

A Flight Guidance System (FGS) is a component of the overall Flight Control System (FCS). It compares the measured state of an aircraft (position, speed, and attitude) to the desired state and generates pitch and roll guidance commands to minimize the difference between the measured and desired state. A simplified overview of an FCS that emphasizes the role of the FGS is shown in Figure 10.

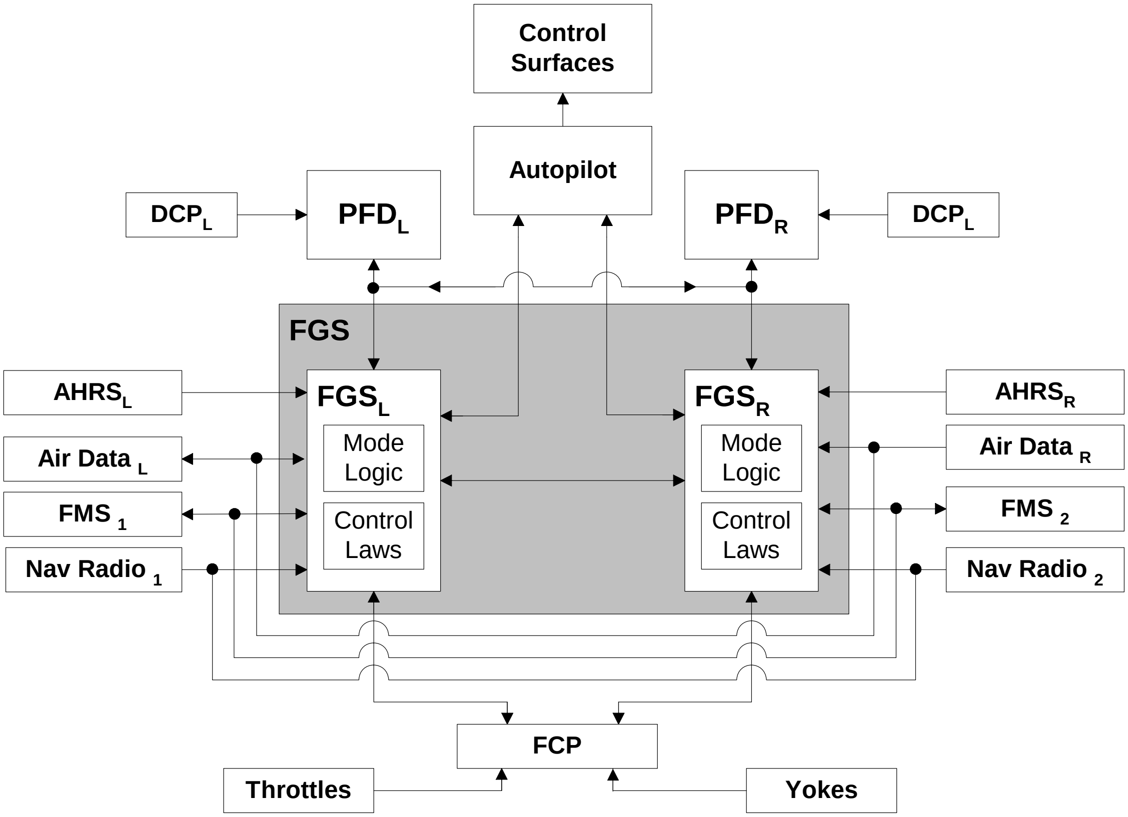

Figure 10 – Flight Control System Overview

As shown in Figure 10, the FGS subsystem accepts input about the aircraft's state from the Attitude Heading Reference System (AHRS), Air Data System (ADS), Flight Management System (FMS), and Navigation Radios. Using this information, it computes pitch and roll guidance commands that are provided to the autopilot (AP). When engaged, the autopilot translates these commands into movement of the aircraft's control surfaces necessary to achieve the commanded changes about the lateral and vertical axes.

The flight crew interacts with the FGS primarily through the Flight Control Panel (FCP), shown in more detail in Figure 11. The FCP includes switches for turning the Flight Director (FD) on and off, switches for selecting the different flight modes such as vertical speed (VS), lateral navigation (NAV), heading select (HDG), altitude hold (ALT), and approach (APPR), the Vertical Speed/Pitch Wheel, and the autopilot disconnect bar. The FCP also supplies feedback to the crew, indicating selected modes by lighting lamps on either side of a selected mode's button. Figure 11 depicts a configuration in which Heading Select (HDG) mode is selected.

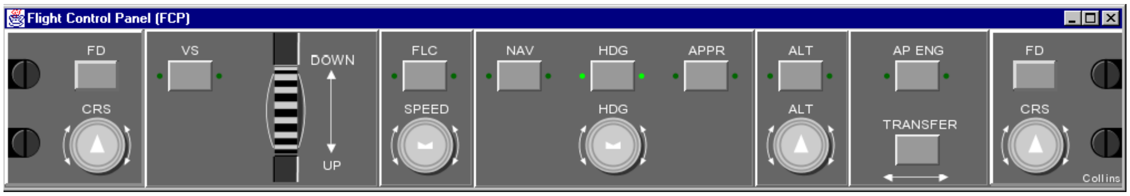

Figure 11 - Flight Control Panel

A few key controls, such as the Go Around button and the autopilot Disengage switch, are provided on the control yokes and throttles and routed through the FCP to the FGS. Navigation sources are selected through the Display Control Panel (DCP), with the selected navigation source routed through the PFD to the FGS.

The FGS has two physical sides, or channels, one on the left side and one on the right side of the aircraft (see Figure 10). These provide redundant implementations that communicate with each other over a cross-channel bus. Each channel of the FGS can be further broken down into the mode logic and the flight control laws. The flight control laws accept information about the aircraft's current and desired state and compute the pitch and roll guidance commands.

The mode logic determines which lateral and vertical modes of operation are active and armed at any given time. These in turn determine which flight control laws are active and armed. These are annunciated, or displayed, on the Primary Flight Displays (PFD) along with a graphical depiction of the flight guidance commands generated by the FGS. A simplified image of a Primary Flight Display (PFD) is shown in Figure 12.

The PFDs display essential information about the aircraft, such as airspeed, vertical speed, attitude, the horizon, and heading. The active lateral and vertical modes are displayed (annunciated) at the top of the display. The annunciations in Figure 12 indicate that the current active lateral mode is Heading Select (HDG), the active vertical mode is Pitch (PTCH), and that Altitude Select (ALTS) mode is armed.

The large sphere in the center of the PFD is the sky/groundball. The horizontal line across its middle is the artificial horizon. The current pitch and roll of the aircraft is indicated by a white wedge ^ representing the aircraft in the middle of the sky/ground ball. Figure 12 depicts an aircraft with zero degrees of roll and pitched up approximately five degrees.

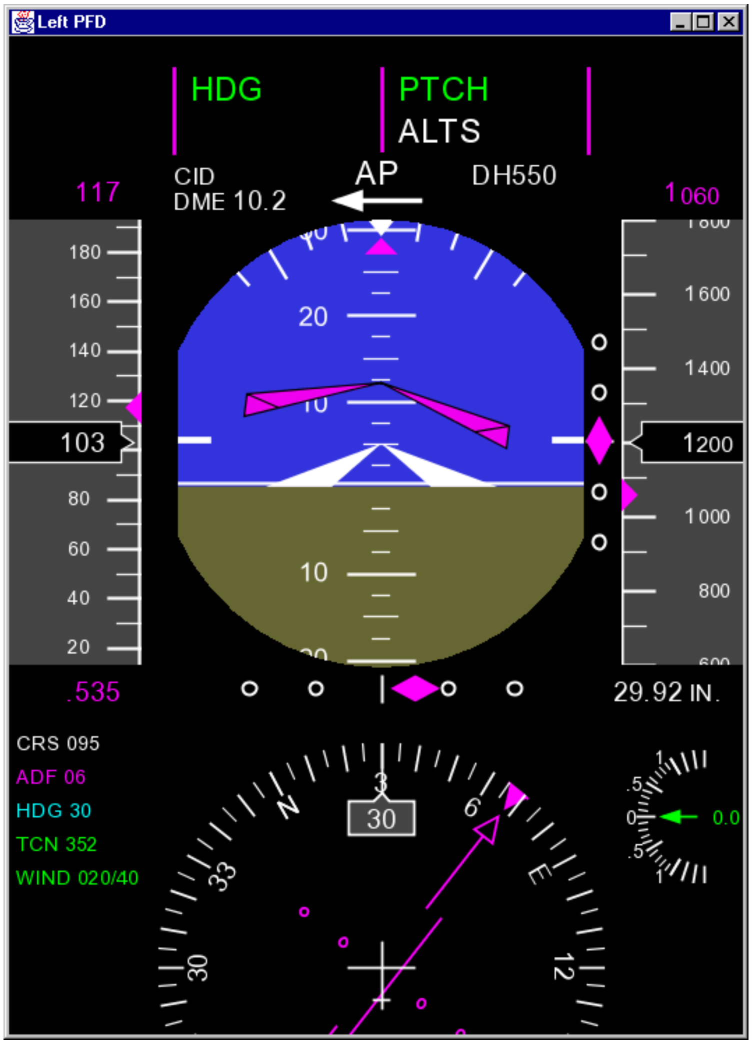

Figure 12 - Primary Flight Display

The graphical presentation of the pitch and roll guidance commands on the PFD are referred to as the Flight Director (FD).[3] The pitch and roll guidance commands are shown as a magenta wedge ^ in the sky/ground ball. When the autopilot is not engaged, these are interpreted as guidance to the pilot. When the autopilot is engaged, these indicate the direction the aircraft is being steered by the autopilot. Figure 12 depicts an aircraft in which the autopilot is not engaged and the Flight Director is commanding the pilot to pitch up and roll to the right.

[3] The term Flight Director is also commonly used to refer to the logic that computes the pitch and roll guidance commands.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*
01- 04 - 2006

2. REPORT TYPE
Contractor Report

3. DATES COVERED *(From - To)*
11/2004 - 11/2005

4. TITLE AND SUBTITLE
Testing Strategies for Model-Based Development

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Heimdahl, Mats P.E.; Whalen, Michael W.; Rajan, Ajitha;
Miller, Steven P.

5d. PROJECT NUMBER
NCC-1-01001

5e. TASK NUMBER

5f. WORK UNIT NUMBER
609866.02.07.07

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)
NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
NASA/CR-2006-214307

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 61
Availability: NASA CASI (301) 621-0390

13. SUPPLEMENTARY NOTES
Langley Technical Monitor: Ricky W. Butler
An electronic version can be found at http://ntrs.nasa.gov

14. ABSTRACT

This report presents an approach for testing artifacts generated in a model-based development process. This approach divides the traditional testing process into two parts: requirements-based testing (validation testing) which determines whether the model implements the high-level requirements and model-based testing (conformance testing) which determines whether the code generated from a model is behaviorally equivalent to the model. The goals of the two processes differ significantly and this report explores suitable testing metrics and automation strategies for each. To support requirements-based testing, we define novel objective requirements coverage metrics similar to existing specification and code coverage metrics. For model-based testing, we briefly describe automation strategies and examine the fault-finding capability of different structural coverage metrics using tests automatically generated from the model.

15. SUBJECT TERMS

Testing; Requirements; Model-based development; Automated test case generation; Requirements-based testing; Test coverage; MC/DC; Unique first cause; UFC; DO-178B

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STI Help Desk (email: help@sti.nasa.gov)
					19b. TELEPHONE NUMBER *(Include area code)*
U	U	U	UU	69	(301) 621-0390