UNIVERSITY OF MINNESOTA

This is to certify that I have examined this copy of a doctoral thesis by

Ajitha Rajan

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Dr. Mats P.E. Heimdahl
Name of Faculty Adviser(s)

Signature of Faculty Adviser(s)

Date

GRADUATE SCHOOL

Coverage Metrics for Requirements-Based Testing

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Ajitha Rajan

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Dr. Mats P.E. Heimdahl, Advisor
August, 2009

Acknowledgments

I am greatly indebted to my advisor, Prof. Mats Heimdahl, for everything that I learned during the course of my PhD. He has been an excellent guide and a huge source of motivation. He has been instrumental in helping me see the forest among trees in all our research problems. He is a pillar of support and has constantly helped me feel positive and confident.

I would like to thank my colleague and friend, Dr. Michael Whalen. Without his guidance, I would have been extremely lost and confused during my first year. I would also like to acknowledge his help in coming up with the notion of requirements coverage. The empirical investigations in this dissertation could not have been carried out without Mike's help and support.

I would like to acknowedge Kurt Woodham at L3 Communications, and Dr. Steve Miller from Rockwell Collins Inc. for their advise and insightful comments. I would like to acknowledge Dr.Elizabeth Whalen from Boeing for her help with the statistical analysis in our empirical study.

I would like to thank my friends/colleagues Anjali Joshi and Matt Staats for several interesting discussions and help with the experiments. I would also like to acknowledge my friend, Satish Sivaswamy, for his constant support, and patience with my occasional rants during my PhD.

I would like to thank my parents and my sister for always being there for me. Without their support and encouragement , I would never have embarked on the PhD adventure. I would also like to thank my fiance, Kartic Subr, for motivating me and supporting me through the last year of my PhD.

Dedication

To my parents, and my sister.

Abstract

In software development, validation that the software meets the customer requirements is accomplished through manual inspections and testing. Current practices in software validation rely on the engineering judgment of domain experts to determine whether or not the tests developed for validation adequately exercise the requirements. There is no objective way of determining the adequacy of validation tests. The work in this dissertation tackles this problem by defining objective metrics termed, *requirements coverage metrics*, that helps determine whether the behaviors specified by the requirements have been adequately tested during software validation.

We define coverage metrics directly on the structure of high-level software requirements.These metrics provide objective, implementation-independent measures of how well a validation test suite exercises a set of requirements. We focus on structural coverage criteria on requirements formalized as Linear Temporal Logic (LTL) properties. These criteria can also be used to automatically generate requirements-based test suites (test suites derived directly from requirements) so that the high-cost of manually developing test cases from requirements is reduced. To achieve this, we developed a framework that automates the generation of requirements-based test cases providing requirements coverage . Unlike model or code-derived test cases, these tests are immediately traceable to high-level requirements. We illustrate the usefulness of the proposed metrics and test case generation technique with empirical investigations on realistic examples from the civil avionics domain.

Another potential application of requirements coverage metrics—in the model-based software development domain—is to measure adequacy of conformance test suites. Conformance test suites are test suites developed to test the adherence of an

implementation to its specification. We found that the effectiveness of existing adequacy metrics can be improved when they are combined with requirements coverage metrics. Test suites providing requirements coverage are capable of revealing faults different from test suites providing coverage defined by existing metrics. We support this claim with empirical evidence and statistical analysis illustrating the usefulness of requirements coverage metrics for measuring adequacy of conformance test suites.

To summarize, this dissertation introduces the notion of requirements coverage and defines potential metrics that can be used to assess requirements coverage. We make the following claims, supported by empirical evidence, regarding the usefulness of requirements coverage metrics:

1. Provides an objective measure of adequacy for software validation testing;
2. Allows for autogeneration of tests immediately traceable to requirements;
3. Improves effectiveness of existing measures of adequacy for conformance testing.

Contents

List of Figures

Chapter 1

Introduction

The classic approach to developing software (as shown stylistically in Figure 1.1) begins with customer specification of requirements (description of what the system should do) and analysis of the requirements to identify any inconsistencies or ambiguities. This is typically done by a skilled and experienced software engineer. The development cycle then proceeds to designing the software, and implementing the design. Through the entire software development life cycle, the software is tested and inspected at each stage to ensure the software meets the requirements and the implementation conforms to the design specification. The process of ensuring this is termed as Verification and Validation (V&V). The terms verification and validation are often confused in their usage. Pezze and Young define the terms as; "Assessing the degree to which a software system fulfills its requirements, in the sense of meeting the user's real needs, is called *validation*. *Verification* is checking the consistency of an implementation with a specification, in contrast to validation which compares a description of the system against actual customer needs" [59]. The term "specification" refers to the description of the behavior of the system to be developed, and should not be confused with the term "requirements" that refers to the customer's actual needs. The difference between verification and validation is stated by Boehm [13] as:

Validation: Are we building the right product ?

Verification Are we building the product right ?

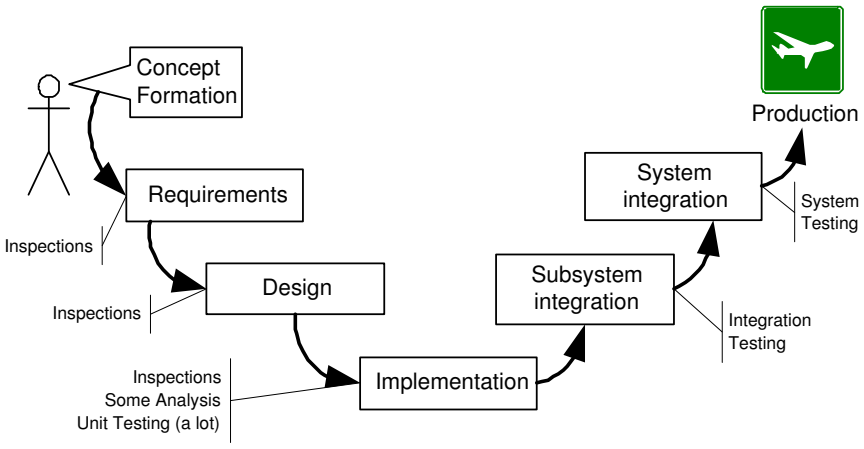

Figure 1.1: Traditional Software Development Process.

In safety-critical systems, systems whose failure may result in loss of life or severe damage to equipment or environment, for example, airplanes, pacemakers, etc., the validation and verification phase (V&V) is particularly costly and consumes a disproportionably large share of the development resources. It is normal for V&V to take up more than 50% of the total development costs for critical systems. Thus, if we could devise techniques to help reduce the cost of V&V, dramatic cost savings could be achieved. The research in this dissertation is an attempt to address this problem.

In particular, we focus on the challenges encountered in software validation. The *software validation* problem—determining that the software accurately captures the customers' high-level requirements—and the sufficiency of the validation activities has been largely determined through ad-hoc methods relying on the engineering judgement of domain experts. In this dissertation, we attempt to provide an objective measure of adequacy of validation activities by answering the following question, *"how well does the system implement the behaviors specified by the requirements?"*

One possible way of addressing this question is through the use of formal verification, i.e., proving or disproving the correctness of a system design with respect

to a desired behavior by checking whether a mathematical representation of the design (formal model) satisfies a specification of this behavior. Formal verification will exhaustively explore all possible behaviors of the system to check the correctness with respect to a desired behavior. Nevertheless, the size and complexity of many industrial systems make the use of formal verification infeasible, even if we have a formal model of the system and formalized requirements. Exponential increase in the number of system behaviors with system size make formal verification extremely expensive and time consuming and often impossible to use on industrial sized systems. Therefore, in practice, software validation is carried out using other techniques such as manual inspection and validation testing.

The goal in validation testing is to provide assurance that software meets all the customer's requirements. Validation testing is achieved using requirements-based testing, a testing approach that involves deriving test cases for each requirement to demonstrate that the system has properly implemented its requirements. Requirements-based testing is also referred to as *black-box* testing. Black-box tests are designed to adequately exercise the functional requirements of a system without regard to the internal workings of a program. The tests are designed considering only the functional requirements and the system interface. For example, consider a sample requirement that states

```
" If Switch is pressed, then the Light will turn on"
```

An analyst developing requirements-based tests or black-box tests may derive the following test case to demonstrate that the requirement is met:

```
Switch = Pressed, Light = On
```

In the above test case, $Switch = Pressed$ is the input data and the expected output is $Light = On$; knowing the expected response is essential in determining

whether actual program execution passed/failed the test case. Does passing such a test case indicate that the system has correctly implemented the requirement? The above requirement can also be met with a test case that never presses the switch. Testing a requirement usually necessitates several test cases to ensure that all the behaviors specified in the requirement are exercised. Nevertheless, it is not feasible to test every possible behavior specified in the requirement; an activity equivalent to formal verification of the requirements. Therefore, we need a criterion that indicates that the behaviors specified in the requirement have been *adequately*—rather than exhaustively—tested. Such a criterion is referred to as a test adequacy criterion and gives an indication of the thoroughness of testing. A test adequacy criterion imposes a set of requirements referred to as *test obligations* to be satisfied by a test suite (a set of test cases). For example, the statement coverage adequacy criterion defined over the structure of programs is satisfied by a particular test suite for a particular program if each executable statement in the program is executed by at least one test case in the test suite. If a test suite fails to satisfy an adequacy criterion, the obligation that has not been satisfied indicates how the test suite is inadequate. If a test suite satisfies all the obligations imposed by an adequacy criterion, we still do not know definitively that it is a well-designed and effective test suite, but we at least have some indication of its thoroughness.

The question we put forth earlier can be restated in the context of validation testing as, *how can one assess whether or not black-box tests adequately cover the behaviors specified in the requirements?* To answer this question, we define an objective notion termed, *requirements coverage*, that determines whether the behaviors specified by the requirements have been adequately tested. Current practices for measuring black-box test suite adequacy either rely on the engineering judgment of domain experts or the adequacy is inferred by examining different coverage metrics

on an executable artifact, either source code [10, 44] or software models [2, 64]. Example of such coverage metrics defined over the structure of an implementation are statement coverage, decision coverage, condition coverage, etc. Section 2.2.1 defines and discusses these metrics in more detail.

There are several problems with the current practice of using executable artifacts to measure the adequacy of black-box tests. First, it is an indirect measure: if an implementation is missing functionality specified by the requirements, a weak set of black-box tests may yield structural coverage of the implementation and yet not expose defects of omission. Conversely, if black-box tests yield poor coverage of an implementation, an analyst must determine whether it is because (a) there are missing or implicit requirements, (b) there is code in the implementation that is superfluous and is not derived from the requirements, or (c) the set of tests derived from the requirements was inadequate. Finally, an executable artifact is necessary to measure the adequacy of the test suite. This may mean that the adequacy of a test suite cannot be determined until late in the development process.

The requirements coverage metrics defined in this dissertation are desirable because they provide *objective, implementation-independent* measures of how well a black-box test suite exercises a set of requirements. Further, given a set of test cases that achieve a certain level of structural coverage of the high-level requirements, it is possible to measure model or code coverage to objectively assess whether the high-level requirements have been sufficiently defined for the system. To our knowledge, there has been no previous work to determine how validation test suites cover the high-level requirements. Most closely related are manual black-box testing methods. For example, Beizer's techniques [8] provide excellent guidance on how to select effective black-box tests from informal requirements and Richardson et al. defined a detailed method for systematically selecting functional tests (black-box tests) from

informal requirements [69].

Another related issue in the software validation activity is the cost associated with developing the validation test cases. Traditionally, software validation has been largely a manual endeavor wherein developers create requirements-based tests and inspect the model to ensure it satisfies the requirements. This is a costly and time consuming process and we try to address this issue also in this dissertation.

We propose an approach to *auto-generate* requirements-based test cases using the requirements coverage metrics and an executable formal model of the system. The requirements-based test cases thus generated can be used to validate if the requirements are implemented correctly in the system. The benefit in generating tests using our approach is that the generated test cases can be immediately traced back to a high-level requirement of interest. This aspect may be helpful to satisfy the testing guidelines of rigorous development processes such as DO-178B [70]. One significance of this approach is that the requirements-based tests can be created early in the development process and thus be used to eliminate defects early in the development process, saving time and resources. In addition, when the tests are generated, inconsistencies and ambiguities, if any, in the requirements will often be revealed. Therefore, this exercise allows the requirements to be improved before the implementation is developed. An additional benefit of generating requirements-based tests using our proposed approach is that we do not introduce any implementation bias in the generated tests, thus the tests are created in a truly black-box manner.

In sum, the primary cost in software validation lies in *creating* tests to sufficiently test the required software behavior and in *showing* that the tests are adequate for validation of the software system. In this dissertation, we attempt to reduce the software validation cost by addressing the following:

1. Providing a means for *objective measurement of the validation activities* so that we can move away from ad-hoc methods for determining requirements coverage (degree to which the requirements are tested). To achieve this, we define coverage metrics directly on the structure of high-level software requirements and illustrate how these metrics can be used as an objective adequacy measure.

2. Providing a foundation for the automation of requirements-based test case generation (generating test cases directly from requirements) so that the high-cost of manually developing test cases from the requirements can be reduced.

Coverage metrics have been proposed in the past as an adequacy criterion for testing. Nevertheless, there has been little work in evaluating the effectiveness of the proposed metrics. Evaluation is crucial if proposed metrics are to be considered in industrial practice and the lack of such evaluation is disturbing. We overcome this drawback-at least partially-in our ongoing work. We evaluate the usefulness of the proposed metrics and test case generation mechanisms with empirical investigations on realistic examples from the civil avionics domain.

In many of the safety critical system domains, including civil avionics, there has been a move away from the traditional view of software development to a new paradigm called *model-based development*. The industrial systems used in our empirical investigations were developed using the model-based software development approach. We present this approach in the next section and discuss the applications of requirements coverage metrics in the context of model-based software development.

1.1 Testing in Model-Based Software Development

In model-based development, the development effort is centered on a formal description of the proposed software system—the "model". This model is derived from

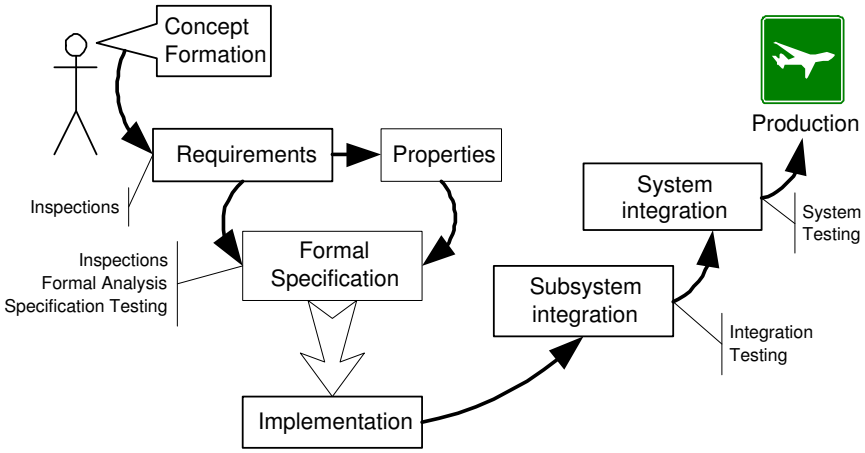

Figure 1.2: Specification Centered Development Process.

high-level requirements describing the expected behavior of the software. For validation and verification purposes, this model can then be subjected to various types of analysis, for example, completeness and consistency analysis [35, 38], model checking [21, 29, 16, 20, 11], theorem proving [5, 9], and test case generation [15, 3, 27, 24, 12, 57, 41]. This shift towards model-based development naturally leads to changes in the verification and validation (V&V) process — V&V has been largely moved from testing the code (Figure 1.1) to analyzing and testing the model (Figure 1.2). The model is also referred to as the specification. The traditional testing process will in a model-based world be split into two distinct activities: *model validation testing* and *conformance-testing*. Figure 1.3 shows an overview of the different testing activities within a model-based development environment. Model validation testing *validates* that the model accurately captures the behavior we really want from the software, i.e., we test the model to convince ourselves that it satisfies the customers' high-level requirements. Conformance testing *verifies* that the code developed from the model (either manually or automatically through code-generators) correctly implements the behavior of the model, i.e., that the implementation conforms to the model and no

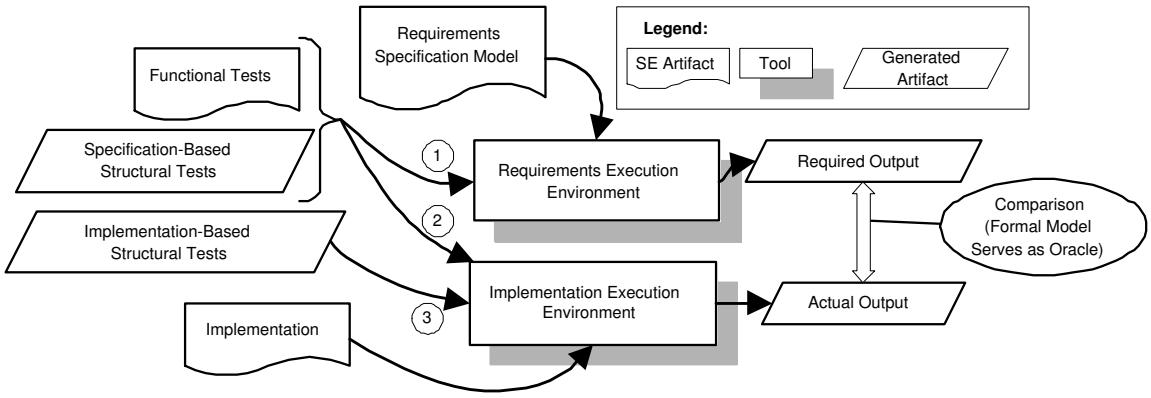

Figure 1.3: An overview of the specification centered testing approach.

"bugs" were introduced in the coding stage or by the automated code generator. Note here that these testing activities are different than what is commonly referred to as model-based testing. In model-based testing, the model is typically constructed specifically for the testing activities and is generally an incomplete abstraction of the system behavior. Naturally, correctness of the model is important, but it does not serve as central of a role as it does in our application domain of model-based development. In model-based development, the model is a complete (or very close to complete) description of the desired behavior from which production code can be directly derived.

Currently, to accomplish the model validation activity in model-based development a set of requirements-based functional tests is manually developed from the informal high-level requirements to evaluate the required functionality of the model. The tests used in this step are developed from the informal requirements by domain experts, much like requirements-based tests in a traditional software development process (Step 1 in Figure 1.3). Validation issues raised in the traditional software development process remain a concern in model-based development also, although in the context of the model. The same key validation questions are raised (1) when

have we developed enough requirements-based tests to ensure the behaviors specified by the requirements are adequately exercised? and (2) can the process of creating requirements-based tests be automated to some extent? We propose to use the requirements coverage metrics to assess the adequacy of the model validation testing activity. Additionally, we provide a test generation framework to automatically generate requirements-based test cases for model validation.

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. In conformance-testing, we are interested in determining whether the implementation correctly implements the formal model. In this stage, the formal model is now assumed to be correct and is used as an oracle. The testing activity confirms that for all tests the implementation yields the same result as the formal model. All tests used during the validation testing of the formal specification can naturally be reused when testing the implementation (step 2 in Figure 1.3). The test cases developed to test the model provide the foundation for testing the implementation, and the executable model serves as an oracle during the testing of the implementation. Again, this test set may not provide adequate coverage of the implementation and will most likely have to be augmented with additional test cases (Step 3 in Figure 1.3).

The issues in *conformance-testing* have received more attention than model validation testing; it is possible to automate the generation of conformance-tests from the formal models as well as the execution of these test cases [66, 68, 12, 27]. Unfortunately, this body of work has largely investigated whether it is *possible* to automate these activities and how to feasibly generate the tests. Very little work has been expended on determining how much to test and if this automation is *effective* in terms of fault finding [37, 34, 73]. For critical avionics software, DO-178B necessitates test

cases used in verification to achieve requirements coverage in addition to structural coverage over the code. Presently, however, owing to the lack of a direct and objective measure of requirements coverage, adequacy of tests are instead inferred by examining structural coverage achieved over the model.Unfortunately, the evidence of using structural coverage over the model to measure the adequacy of conformance testing in the model-based domain is highly inconclusive, and some serious concerns as to its effectiveness is raised in [62]. Preliminary studies [34, 33] have raised serious doubts about the fault-finding capability of the various model (or specification) coverage criteria suggested in the literature.

We hypothesize that adequacy metrics for conformance testing should consider *structural coverage over requirements* either in place of or in addition to structural coverage over the model. Measuring structural coverage over the requirements gives a notion of how well behaviors specified in the requirements are implemented in the code. We propose to use the requirements coverage metrics defined in this dissertation to help achieve this. We conduct empirical studies to investigate this hypothesis and evaluate the effectiveness of the test sets based on their fault finding capability in the implementation. Chapter 5 presents this empirical investigation.

In sum, in the domain of model-based software development, we propose to use requirements coverage metrics as an objective means of measuring adequacy of model validation testing and as a basis for auto generating model validation test suites. We also believe that the requirements coverage metrics will be useful in measuring adequacy of conformance test suites.

1.2 Contributions

Thus far we have discussed the challenges in V&V in both the traditional and model-based software development approaches. We propose to define requirements coverage

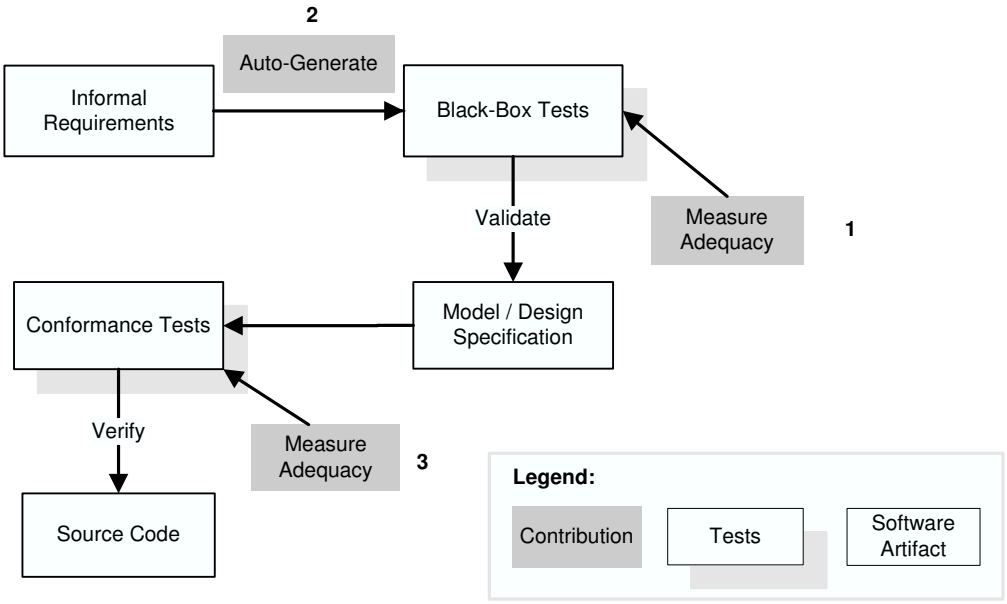

Figure 1.4: Dissertation Contributions in Software Development.

metrics to address some of these challenges. The contributions relevant to V&V of software provided by this dissertation are illustrated in Figure 1.4 and summarized below. Note that the numbers 1, 2, and 3 in Figure 1.4 correspond to the contributions enumerated below.

1. A collection of requirements adequacy criteria that can be used as a *direct measure* of how well a black-box test suite addresses a set of requirements. The requirements adequacy criteria also provide an *implementation independent* assessment of the adequacy of a suite of black-box tests used in validation. The collection of requirements adequacy criteria will have been formally defined, evaluated, and compared with respect to their effectiveness in achieving coverage over implementation, in the domain of safety-critical reactive systems.

2. Development of a formal framework that allows for *auto-generation of validation tests* that are immediately traceable to requirements. Provide empirical

evidence on the feasibility and usefulness of the requirements adequacy criteria used for the purpose of requirements-based test case generation.

3. Provide empirical evidence on the usefulness of requirements coverage metrics for measuring adequacy of conformance test suites used in model-based development. We evaluate usefulness of the metrics in terms of their improvement in fault finding over existing metrics.

1.3 How to Read this Dissertation

In Chapter 2, we provide background information on software testing and existing metrics to measure adequacy of testing. To better understand the coverage metrics we defined on formalized requirements, we also provide some background on how to formalize requirements. Readers familiar with these topics can skip forward to Chapter 3.

Chapter 3 presents the metrics we defined for measuring coverage over requirements formalized as LTL properties. We formally define and discuss three requirements coverage metrics. We discuss how to adapt the definitions of the metrics to measure finite test cases. We also present an empirical study that compares the effectiveness of the defined requirements coverage metrics on a close to production model of a flight guidance system. Section 3.3 presents the requirements coverage measurement tool that provides the capability to measure requirements coverage achieved by a test suite on requirements formalized as LTL properties.

We describe our proposed approach to automatically generate requirements-based test cases for use in validation testing in Chapter 4. The chapter also presents an empirical study that evaluates the effectiveness of the generated requirements-based tests on realistic examples.

In Chapter 5, we illustrate that a requirements coverage metric discussed in Chapter 3 is useful in measuring adequacy of conformance test suites. We present an empirical study using realistic examples that investigates the effectiveness of requirements coverage against an existing adequacy measure for conformance testing in the critical systems domain.

In Chapter 6, we present literature related to defining requirements coverage metrics and their use as a test adequacy metric. We also present literature related to requirements-based testing.

Finally, Chapter 7 summarizes the work in this dissertation and points to directions for future work. To understand our approach to automate the generation of requirements-based test cases, we present notations for formal modeling and existing tools for automated test case generation in Appendix A

Chapter 2

Background

We first present a brief discussion of what software testing means and the various types of testing. We then discuss how adequacy of black-box test suites is currently measured. Related to this we outline the structural coverage metrics that have been defined over implementations. To better understand our definition of coverage over formal high-level requirements, we present a discussion about requirements in a model-based world and how to formalize them.

2.1 Software Testing

Software testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required behavior [39]. In spite of the importance of software testing, it still remains an art due to limited understanding of the principles of software. The difficulty in software testing stems from the complexity of software: we can not completely test a program with even moderate complexity. The purpose of testing can be quality assurance, verification and validation, or reliability estimation. Software testing is a trade-off between budget, time and quality.

The tester may or may not know the inside details of the software module under test, e.g. control flow, data flow, etc. The two basic techniques of software testing, *black-box testing* and *white-box testing* are explained as follows:

Black-box testing: The black-box approach is a testing method in which test cases are derived from the specified functional requirements without regard to the final program structure [58]. It is also termed data-driven, input/output driven, or requirements-based [39] testing. Since only the functionality of the software module is of concern, black-box testing also mainly refers to functional testing, a testing method that emphasizes on executing the functions and examining their input and output data. The tester treats the software under test as a black-box—only the inputs, outputs and specification are visible—and the functionality is determined by observing the outputs to corresponding inputs. In testing, various inputs are exercised and the outputs are compared against the specification to validate the correctness. All test cases are derived from the specification. No implementation details of the code are considered.

White-box testing: In this testing technique, software is viewed as a white-box, or glass-box, as the structure and flow of the software under test are visible to the tester. Testing plans are made according to the details of the software implementation, such as programming language, logic, and styles. Test cases are derived from the program structure. White-box testing is also called logic-driven testing or design-based testing [39].

2.2 Adequacy of Black-Box Test Suites

We first look at what an adequacy criterion for a test suite implies. According to Pezze and Young [59], a test suite satisfies an adequacy criterion if all the tests succeed and if every test obligation in the criterion is satisfied by at least one of the test cases in the test suite. For example, the statement coverage adequacy criterion is satisfied by a particular test suite for a particular program if each executable statement in

the program is executed by at least one test case in the test suite and the actual results from the executions match the expected results. If a test suite fails to satisfy some criterion, the obligation that has not been satisfied may provide some useful information about improving the test suite. If a test suite satisfies all the obligations by all the criteria, we still do not know definitively that it is a well-designed and effective test suite, but we have at least some indication of its thoroughness.

Presently, the adequacy of black-box test suites is inferred by examining different coverage metrics on an executable artifact. As mentioned earlier, there are several problems with using the executable artifacts to measure the adequacy of black-box tests. In the following section we look at some of the existing metrics of adequacy defined over the structure of an executable artifact.

2.2.1 Structural Coverage Criteria over Implementation

Structural coverage criteria defined over implementations refer to structural coverage criteria defined over source code or software models. Code coverage is based on code, for example C-code or Java, and model coverage is based on graphical models such as Simulink models or UML models. According to Chilenski and Miller [44], structural coverage criteria are divided into two types: data flow and control flow. Data flow criteria measure the flow of data between variable assignments and references to the variables. Data flow metrics, such as all-definitions and all-uses [10], involve analysis of the paths (or subpaths) between the definition of a variable and its subsequent use. The structural coverage criteria in many standards, including DO-178B, are often control flow criteria. Control flow criteria measure the flow of control between statements and sequences of statements. For control flow criteria, the degree of structural coverage achieved is measured in terms of statement invocations, Boolean expressions evaluated, and control constructs exercised. Descriptions of some of the

commonly used structural coverage measures based on control flow, as defined by Hayhurst et al. in [32], are as follows:

Statement Coverage: To achieve statement coverage, every executable statement in the program is invoked at least once during software testing. Achieving statement coverage shows that all code statements are reachable. Statement coverage is considered a weak criterion because it is insensitive to some control structures.

Decision Coverage: Decision coverage requires two test cases: one for a true outcome and another for a false outcome. For simple decisions (i.e., decisions with a single condition), decision coverage ensures complete testing of control constructs. However, not all decisions are simple. For the decision (A *or* B), test cases (TF) and (FF) will toggle the decision outcome between true and false. However, the effect of B is not tested; that is, those test cases cannot distinguish between the decision (A *or* B) and the decision A.

Condition Coverage: Condition coverage requires that each condition in a decision take on all possible outcomes at least once (to overcome the problem in the previous example), but does not require that the decision take on all possible outcomes at least once. In this case, for the decision (A *or* B) test cases (TF) and (FT) meet the coverage criterion, but do not cause the decision to take on all possible outcomes. As with decision coverage, a minimum of two tests cases is required for each decision.

Condition/Decision Coverage: Condition/decision coverage combines the requirements for decision coverage with those for condition coverage. That is, there must be sufficient test cases to toggle the decision outcome between true and

false and to toggle each condition value between true and false. Hence, a minimum of two test cases are necessary for each decision. Using the example (A *or* B), test cases (TT) and (FF) would meet the coverage requirement. However, these two tests do not distinguish the correct expression (A *or* B) from the expression A or from the expression B or from the expression (A *and* B).

Modified Condition/Decision Coverage: The MC/DC criterion enhances the condition/decision coverage criterion by requiring that each condition be shown to independently affect the outcome of the decision. The independence requirement ensures that the effect of each condition is tested relative to the other conditions. However, achieving MC/DC requires more thoughtful selection of the test cases, and, in general, a minimum of n+1 test cases for a decision with n inputs. For the example (A *or* B), test cases (TF), (FT), and (FF) provide MC/DC. For decisions with a large number of inputs, MC/DC requires considerably more test cases than any of the coverage measures discussed above.

Multiple Condition Coverage: Finally, multiple condition coverage requires test cases that ensure each possible combination of inputs to a decision is executed at least once; that is, multiple condition coverage requires exhaustive testing of the input combinations to a decision. In theory, multiple condition coverage is the most desirable structural coverage measure; but, it is impractical for many cases. For a decision with n inputs, multiple condition coverage requires 2^n tests.

2.3 Requirements in a Model-Based World

We propose to define coverage metrics directly on the structure of requirements as measures of adequacy for validation test suites. In order to define this coverage

over requirements written in natural language, we would first need to express the requirements in a formal, unambiguous way. In this section we discuss how natural language requirements can be formalized. This section is largely adapted from Section 2 in a NASA Contractor Report produced in the context of this research [76].

Requirements typically begin in the form of natural language statements. The natural language requirements are usually expressed as "shall" statements [71]. The flexibility and expressiveness of natural language, which are so important for human communications, represent an obstacle to automatic analysis. We therefore use properties (or formalized requirements) captured using formal notations such as temporal logics [22] and synchronous observers [30]. These properties are very close in structure to the "shall" statements making up the requirements; in fact, [53] argues that the "shall" requirements are simply desirable properties of our system expressed in an informal and ambiguous notation. It is worth noting that the properties referred to here are different from the term "model" used in this dissertation. A model as mentioned in Section 1.1 is a formal description of the proposed software system and is an abstract implementation of the properties. Some of the notations used to model a system are described in the Appendix.

In our work, we focus on structural coverage metrics defined over requirements formalized in the LTL notation. Briefly, LTL is a notation that allows you to represent and reason about formulae with temporal constraints over paths. For instance, one can encode an LTL formulae about the future of paths such as that a condition will eventually be true, or that a condition will be true until another condition becomes true, etc. The temporal operators in LTL are described in Table 2.1.

Generally, there is not a one-to-one relationship between many of the natural language requirements and the formal properties. One requirement may lead to a number of properties. We introduce here a Flight Guidance System (FGS) example

Operator	**Notation**	**Meaning**
Globally ϕ	$G\ \phi$	Globally ϕ has to hold on the entire subsequent path
Future ϕ	$F\ \phi$	Eventually ϕ has to hold
Next ϕ	$X\ \phi$	ϕ has to hold in the next state
ϕ Until ψ	$\phi\ U \psi$	ϕ has to hold in every state until ψ holds. ψ must eventually hold

Table 2.1: Summary of the Linear Time Temporal Logic (LTL) operators.

to illustrate this one–to–many requirement–property relationship. A more detailed description is available in [76].

The 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. 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 system to possess. The number of properties depends on which lateral modes are included in

this particular configuration of the flight guidance system. For example, one version of the FGS used in our study has five lateral modes (ROLL, HDG, NAV, LGA, and LAPPR) leading to the following properties.

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.

1. $G(Is\_ROLL\_Active \rightarrow (\neg Is\_HDG\_Active \wedge \neg Is\_NAV\_Active \wedge \neg Is\_LGA\_Active \wedge \neg Is\_LAPPR\_Active))$
2. $G(Is\_HDG\_Active \rightarrow (\neg Is\_ROLL\_Active \wedge \neg Is\_NAV\_Active \wedge \neg Is\_LGA\_Active \wedge \neg Is\_LAPPR\_Active))$
3. $G(Is\_NAV\_Active \rightarrow (\neg Is\_ROLL\_Active \wedge \neg Is\_HDG\_Active \wedge \neg Is\_LGA\_Active \wedge \neg Is\_LAPPR\_Active))$
4. $G(Is\_LGA\_Active \rightarrow (\neg Is\_ROLL\_Active \wedge \neg Is\_HDG\_Active \wedge \neg Is\_NAV\_Active \wedge \neg Is\_LAPPR\_Active))$
5. $G(Is\_LAPPR\_Active \rightarrow (\neg Is\_ROLL\_Active \wedge \neg Is\_HDG\_Active \wedge \neg Is\_NAV\_Active \wedge \neg Is\_LGA\_Active))$

On the other hand, some requirements, for example, the requirement in Table 2.2 (also from the FGS), lend themselves directly to a one-to-one formalization. (FD refers to the Flight Director and AP refers to the Auto Pilot.) 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 natural language requirements as properties typically involves

"If the onside FD cues are off, the onside FD cues shall be displayed when the AP is engaged." (a)
$G((\neg Onside\_FD\_On \wedge \neg Is\_AP\_Engaged) \rightarrow X(Is\_AP\_Engaged \rightarrow Onside\_FD\_On))$ (b)

Table 2.2: (a) Sample requirement on the FGS (b) LTL property for the requirement

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. Nevertheless, in most cases—like the example above—the LTL property is very similar in structure to the natural language requirement making the translation straightforward. In [53], the full set of natural language requirements for the FGS were translated into properties in Linear Time Temporal Logic (LTL) [22].

Chapter 3

Requirements Coverage

As mentioned in Section 2.3, there is a close relationship between the requirements and the properties captured for verification purposes. An analyst developing test cases from the natural language requirements (shown in Table 2.2) might derive the scenario in Table 3.1 to demonstrate that the requirement is met. Does this adequately cover this requirement? Does passing such a test case indicate that the system has correctly 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 validation activities. The requirements coverage criteria presented in this chapter is a summary and extension of our work published in [77].

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

Table 3.1: Manually developed requirements-based test scenario

As discussed in Section 2.2.1, several different structural coverage criteria have been investigated for source code and for executable modeling languages, for example,[69,

66, 57, 27]. It is possible to adapt some of these criteria to fit in the domain of requirements captured as formal properties in, for example, Computational Tree Logic (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 requirements, it would require a single test case that demonstrates that the requirement is satisfied by the system—a test case such as the manually developed one in Table 3.1. If we derived such a test case for each requirement, we could claim that we have achieved requirements decision coverage. A distinction must be made, however, between 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.

Recall that the 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, requirements MC/DC coverage. The notion of structural coverage over property syntax will be discussed in detail in Section 3.1.

Although we in this dissertation focus on requirements specified in temporal logic there are other notations that can be used to describe requirements. For example, SCADE [17] and Reactis [48] use synchronous observers [24], which are small specifications of requirements written as state machines or in the same notation as the software specification that run "in parallel" with the model. Structural coverage could be defined over such observers. Nevertheless, we will focus our work on requirements

formalized in some declarative notation (such as LTL) since these formalizations tend to closely resemble the "shall" requirements we find in practice.

3.1 Metrics for Requirements Coverage

In this section we define and discuss three coverage metrics over the structure of requirements formalized as Linear Temporal Logic (LTL) properties. The coverage metrics provide different levels of rigor for measuring adequacy of validation activities, and—depending on the thoroughness of testing needed—the appropriate metric can be chosen. .

3.1.1 Requirements Coverage

To satisfy *requirements coverage*, a test suite should contain at a minimum one test per requirement that illustrates one way in which the requirement can be met. As an example, consider a sample requirement in Table 3.2 (simplified from the one in Table 2.2).

"The onside Flight Director cues shall be displayed when the Auto-Pilot is engaged." (a)
$G(Is\_AP\_Engaged \rightarrow Onside\_FD\_On)$ (b)

Table 3.2: (a) Sample requirement (b) LTL property for the requirement

A test derived from the natural language requirements might look like the following scenario: (1) Engage the AP, and (2) Verify that the Onside FD comes on.

Alternatively, we could simply leave the auto-pilot turned off and it does not matter what happens to the flight director. Technically, this test also demonstrates one way in which the requirement is met, but the test is not particularly illuminating. This metric is a very weak measure of whether we have sufficiently exercised requirements, since there may be several ways of coming up with a test case that appear to address the requirement but are not otherwise helpful, such as the above example that simply leaves the auto-pilot off.

3.1.2 Requirements Antecedent Coverage

Requirements are often of the form "if event A happens, then event B shall happen". This requirement, can be formalized in LTL as

$$G(A \rightarrow B)$$

Informally, it is always the case that when A holds B will hold. A on the left hand side of $\rightarrow$ is referred to as the antecedent, and B on the right hand side as the consequent. The example in Table 3.2 is a requirement of the above form. When using tools for test case generation, the tools will generally find the simplest possible test case that satisfies the property. For requirements of the form mentioned above, the simplest way of satisfying it is when "event A" never happens, or the autopilot is always off in the case of the example in Table 3.2. Such test cases are not particularly interesting and do not exercise the interesting scenario (when event A happens, does event B happen?).

In order to ensure that requirements of the form mentioned are not tested in a naïve and uninteresting manner, we defined *requirements antecedent coverage*. For a test case to provide requirements antecedent coverage over a requirement, the test case should satisfy the requirement with the additional constraint that the antecedent

is exercised. To illustrate this, consider again the example LTL requirement,

$$G(A \rightarrow B)$$

A test case providing requirements antecedent coverage over such a requirement will ensure that the antecedent A becomes true at least once along the satisfying path. That is, the test case would satisfy the obligation

$$G(A \rightarrow B) \wedge F(A)$$

For the requirement in Table 3.2, a test case satisfying requirements antecedent coverage would have to be an execution path where the autopilot is engaged at least once in addition to satisfying the overall requirement. Table 3.3 shows the obligation expressed as an LTL formula, and Table 3.4 shows a sample test case satisfying the obligation (the antecedent is exercised in the third step in the test case).

Formal LTL Requirement: $G(Is\_AP\_Engaged \rightarrow Onside\_FD\_On)$
Requirements Antecedent Coverage:
1. $G(Is\_AP\_Engaged \rightarrow Onside\_FD\_On) \ \wedge \ F(Is\_AP\_Engaged)$

Table 3.3: Example LTL property and obligation for Requirements Antecedent Coverage

Step	**1**	**2**	**3**	**4**
AP	Disengaged	Disengaged	*Engaged*	Disengaged
FD	Off	On	*On*	Off

Table 3.4: Sample Test Case satisfying Requirements Antecedent Coverage

Note that in the illustrated examples, there is only one implication operator ($\rightarrow$). Requirements may be defined with several implication operators (like the example in

Table 2.2). For such requirements, requirements antecedent coverage could be defined to exercise the antecedent in all the implication operators, or simply the antecedent in the top-level implication operator. The former notion over all implication operators is more rigorous. Table 3.5 shows the obligations for the two different notions of requirements antecedent coverage for an example requirement. Presently, the tool we built to measure requirements coverage only considers top-level implication operators. In our future work we plan to include the definition of requirements antecedent coverage that considers all implication operators.

Formal LTL Requirement: $G(A \rightarrow (B \rightarrow C))$
All Implication Operators
Obligation for Antecedent A: $G(A \rightarrow (B \rightarrow C)) \wedge F(A)$
Obligation for Antecedent B: $G(A \rightarrow (B \rightarrow C)) \wedge F(A \wedge B)$
Top-Level Implication Operator
Obligation for Antecedent A: $G(A \rightarrow (B \rightarrow C)) \wedge F(A)$

Table 3.5: Example to illustrate two notions of Requirements Antecedent Coverage

3.1.3 Unique First Cause (UFC) Coverage

Requirements are not usually as simple as the example shown in Table 3.2. They are often defined using complex conditions, like the example FGS requirement below:

> *"If this side is active the mode annunciations shall be on if and only if the onside FD cues are displayed, or the offside FD cues are displayed, or the AP is engaged."*

Formalized as:

$$G(Is\_This\_Side\_Active = 1 \rightarrow (Mode\_Annunciations\_On \leftrightarrow (Onside\_FD\_On \vee Offside\_FD\_On = TRUE \vee Is\_AP\_Engaged)))$$

For such complex requirements (often even simple ones), it can be desirable to have a rigorous coverage metric that requires tests to demonstrate the effect of *each* atomic condition making up the complex conditions in the requirement; this would ensure that every atomic condition is necessary and affects the outcome of the property. The requirements and requirements antecedent coverage defined previously would not be able to do this. Therefore, we define a coverage metric called *Unique First Cause (UFC)* coverage over requirements [77]. It is adapted from the Modified Condition/Decision Coverage (MC/DC) criterion [44, 32] defined over source code. MC/DC is a structural coverage metric that is designed to demonstrate the independent effect of basic Boolean conditions (i.e., subexpressions with no logical operators) on the Boolean decision (expression) 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.

The process of satisfying MC/DC involves determining whether each of the constraints imposed by MC/DC is satisfied by some *state* that is reached by a test within a test suite. Since requirements captured as LTL properties define *paths* rather than states, we broaden our view of structural coverage to accommodate satisfying paths rather than satisfying states. 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. We can define these test 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 $\pi = s_0 \rightarrow s_1 \rightarrow \ldots$ in which the formula holds for states $s_0 \ldots s_{k-1}$, then passes through state s_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 UFC coverage 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 π, an atom α in A is the unique first cause if, in the first state along π in which A is satisfied (i.e. the truth value of formual A is *true*), it is satisfied because of atom α. To make this notion concrete, suppose we have the formula $\mathrm{F}(a \vee b)$ and a path $P = s_0 \rightarrow s_1 \rightarrow \ldots$ in which a was initially true in step s_2 and b was true in step s_5. For path P, a (but not b) would satisfy the unique first cause obligation.

Based on this definition for UFC, Table 3.6 gives the UFC obligations for an example LTL requirement. The discussion included above gives the general idea of the notion of UFC coverage; we next present the complete formalization of UFC coverage over requirements expressed as LTL properties.

Formalization of UFC Coverage over LTL formulas

It is straightforward to describe the set of required UFC assignments for a decision as a set of Boolean expressions. Each expression is designed to show whether a particular

Formal LTL Requirement: $G(Is\_AP\_Engaged \rightarrow Onside\_FD\_On)$

Requirements UFC Coverage

1. $(Is\_AP\_Engaged \rightarrow Onside\_FD\_On)\ U\ ((\neg Is\_AP\_Engaged \wedge \neg Onside\_FD\_On) \wedge\ G(Is\_AP\_Engaged \rightarrow Onside\_FD\_On))$

2. $(Is\_AP\_Engaged \rightarrow Onside\_FD\_On)\ U\ ((Is\_AP\_Engaged \wedge Onside\_FD\_On) \wedge\ G(Is\_AP\_Engaged \rightarrow Onside\_FD\_On))$

Table 3.6: Example LTL property and obligation for requirements UFC coverage

condition positively or negatively affects the outcome of a decision. Note that over expressions with simple boolean operators ($\wedge$, $\vee$, $\neg$), the definition for UFC is the same as the definition for the MC/DC metric over source code and models. 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.

For simple boolean operators, we can define A^+ and A^- schematically over the structure of complex decisions as follows:

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

$x^- = \{\neg x\}$ (where x is a basic condition)

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

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

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

To get positive MC/DC coverage of $A \wedge B$, we need to make sure that every element in A^+ uniquely contributes to making $A \wedge B$ true while holding B true, and symmetrically argue for the elements of B^+. The argument for negative MC/DC coverage is the same, except we show that $A \wedge B$ is false by choosing elements of A^- and B^-.

$$(A \vee B)^+ = \{a \wedge \neg B \mid a \in A^+\} \cup \{\neg A \wedge b \mid b \in B^+\}$$

$$(A \vee B)^- = \{a \wedge \neg B \mid a \in A^-\} \cup \{\neg A \wedge b \mid b \in B^-\}$$

To get positive (negative) MC/DC coverage over $A \vee B$, we need to make sure that every element in A^+ (A^-) uniquely contributes to making $A \vee B$ true (false) while holding B false, and the symmetric argument for elements of B^+ (B^-).

$$(\neg A)^+ = A^-$$

$$(\neg A)^- = A^+$$

The positive and negative MC/DC coverage sets for $\neg A$ swap the positive and negative obligations for A.

Since expressions over simple boolean operators define program or model states, each of the expressions above in the positive and negative sets can be seen as defining a constraint over a program or model state. The process of satisfying UFC involves determining whether each of these constraints is satisfied by some state that is reached by a test within a test suite.

On the other hand, requirements captured as LTL properties define paths rather than states, therefore we broaden our view of structural coverage to accommodate satisfying paths rather than satisfying states. 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. We can define these test paths by extending the constraints for state-based UFC (or MC/DC) to include temporal operators. These operators describe the path constraints required to reach an acceptable state. 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.

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

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

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

$\mathrm{G}(A)$ is true if A is true along all states within a path. The A U $(a \ \wedge \ \mathrm{G}(A))$ 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.

$\mathrm{F}(A)^-$ is the dual of $\mathrm{G}(A)^+$, so the obligations match after negating A and a.

$\mathrm{F}(A)^+ = \{\neg A \text{ U } a \mid a \in A^+\}$

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

The independent effect of $a \in A^+$ for the $\mathrm{F}(A)$ formula is demonstrated by showing that it is the first cause for A to be satisfied. Similar to the previous definition, $\mathrm{G}(A)^-$ is the dual of $F(A)^+$.

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

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

The independent effects of $a \in A^+$ (resp. $a \in A^-$) are demonstrated by showing that they affect the formula in the next state.

$(A \text{ U } B)^+ =$

$\{(A \ \wedge \neg B) \text{ U } ((a \ \wedge \neg B) \wedge (A \text{ U } B)) \mid a \in A^+\} \cup$

$\{(A \ \wedge \neg B) \text{ U } b \mid b \in B^+\}$

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 completing the formula.

The formula to the left of the union provides positive UFC over A in $(A \text{ U } B)$. Recall that an 'until' formula is immediately satisfied if B holds. Therefore, in order to show that some specific atom in A (isolated by the formula 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: $(A \ \wedge \neg B)$ U $(a \ \wedge \neg B)$. In order to ensure that our prefix is sound, we still want B to eventually hold, we add $(A \text{ U } B)$ to complete the formula.

The formula on the right of the union is similar and asserts that some $b \in B$ is the unique first cause for B to be satisfied.

$(A \text{ U } B)^- =$

$\{(A \wedge \neg B) \text{ U } ((a \wedge \neg B) \mid a \in A^-\} \cup$

$\{(A \wedge \neg B) \text{ U } (b \wedge \neg(A \text{ U } B)) \mid b \in B^-\}$

For the formula A U B to be falsified, there is some state in which A is false before B is true. The formula to the left of the union demonstrates that $a \in A^-$ uniquely contributes to the falsehood of the formula 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 both a and B are false.

The formula to the right demonstrates that $b \in B^-$ uniquely contributes to the

falsehood of the formula by describing a path in which A U B eventually fails, but A holds long enough to contain state b falsifying B.

The formulations above define a rigorous notion of requirements 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 apply to finite traces.

3.1.4 Adapting Formulas to Finite Tests

LTL is normally formulated over *infinite* paths while test cases correspond to *finite* paths. Nevertheless, the notions of coverage defined in the previous sections 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 [49, 23, 6], and we can adapt these ideas to check whether a test suite has sufficiently covered a property.

Manna and Pnueli [49] 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. The most significant 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, respectively. The strong operator is always false on the last state in a finite path, while the weak operator is always true.

It is straightforward to define a formal semantics for LTL 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 $s_1 \rightarrow s_2 \rightarrow ... \rightarrow s_k$. 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 3.7. As expected, these definitions correspond with the standard semantics except that

1.	$\pi \vDash true$
2.	$\pi \vDash p$ iff $\lvert\pi\rvert > 0$ and $p \in L(s_1)$
3.	$\pi \vDash \neg A$ iff $\pi \nvDash A$
4.	$\pi \vDash (A \wedge B)$ iff $\pi \vDash A$ and $\pi \vDash B$
5.	$\pi \vDash (A \vee B)$ iff $\pi \vDash A$ or $\pi \vDash B$
6.	$\pi \vDash X(A)$ iff $\lvert\pi\rvert \leq 1$ or $\pi^2 \vDash A$
7.	$\pi \vDash X!(A)$ iff $\lvert\pi\rvert > 1$ and $\pi^2 \vDash A$
8.	$\pi \vDash G(A)$ iff $\forall 1 \leq i \leq \lvert\pi\rvert$ $\pi^i \vDash A$
9.	$\pi \vDash F(A)$ iff for some $1 \leq i \leq \lvert\pi\rvert$ $\pi^i \vDash A$
10.	$\pi \vDash A$ U B iff there is some $1 \leq i \leq \lvert\pi\rvert$ where $\pi^i \vDash B$ and $\forall j = 1..(i-1), \pi^j \vDash A$

Table 3.7: Semantics of LTL over Finite Paths

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 3.7 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:

$$((a \ \vee \ b) \text{ U } c)$$

and the test cases in Table 3.8. 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 $((a \ \vee \ b) \text{ U } c)$ 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

Test1:

Atom	Step1	Step2	Step3
a	t	t	f
b	t	f	t
c	f	f	f

Test2:

Atom	Step1	Step2
a	t	t
b	t	f
c	f	t

Table 3.8: Test Suite for Property $((a \vee b)$ U $c)$

perspective (the perspective of the finite semantics described above), Test 1 does not satisfy the formula (since the finite semantics in Table 3.7 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 by Eisner et al. [23]; they define three different semantics for temporal operators: *weak, neutral,* and *strong*. The *neutral* semantics are the semantics defined by Manna and Pnueli [49] described in Table 3.7. 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.

Since we believe that the test cases in Table 3.8 adequately illustrate the indepen-

dence of a and b, we slightly weaken our LTL obligations. Given a formula α, we are interested in a prefix of an accepting path for α 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*, 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 [23], the semantics are provided as variant reformulations 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. We define $weak[\alpha]$ to be the weakening of a formula α and $strong[\alpha]$ to be the strengthening of formula α. The transformations weak and strong are defined in Table 3.9. We refer the reader to the work by Eisner et al. [23] for a full description of the three semantics and their effect on provability within the defined logic.

1. $weak\ [true] = true$	12. $strong\ [true] = true$
2. $weak\ [p] = p$	13. $strong\ [p] = p$
3. $weak\ [\neg A] = \neg strong\ [A]$	14. $strong\ [\neg A] = \neg\ weak\ [A]$
4. $weak\ [A \wedge B] = weak\ [A] \wedge weak\ [B]$	15. $strong\ [A \wedge B] = strong\ [A] \wedge strong\ [B]$
5. $weak\ [A \vee B] = weak\ [A] \vee weak\ [B]$	16. $strong\ [A \vee B] = strong\ [A] \vee strong\ [B]$
6. $weak\ [X!(A)] = X(weak\ [A])$	17. $strong\ [X!(A)] = X!(strong\ [A])$
7. $weak\ [X(A)] = X(weak\ [A])$	18. $strong\ [X(A)] = X!(strong\ [A])$
8. $weak\ [G(A)] = G(weak\ [A])$	19. $strong\ [G(A)] = false$
9. $weak\ [F(A)] = true$	20. $strong\ [F(A)] = F(strong\ [A])$
10. $weak\ [A\ \mathrm{U}\ B] = weak\ [A]\ W^1 weak\ [B]$	21. $strong\ [A\ U\ B] = strong\ [A]\ \mathrm{U}\ strong\ [B]$
11. $weak\ [A\ \mathrm{W}\ B] = weak\ [A]\ W weak\ [B]$	22. $strong\ [A\ W\ B] = strong\ [A]\ \mathrm{U}\ strong\ [B]$

Table 3.9: 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 UFC over finite prefixes are shown in Table 3.10.

$$
\begin{aligned}
G(A)^+ &= \{A \text{ U } (a \wedge \mathit{weak}[G(A)]) \mid a \in A^+\} \\
G(A)^- &= \{A \text{ U } a \mid a \in A^-\} \\
F(A)^+ &= \{\neg A \text{ U } a \mid a \in A^+\} \\
F(A)^- &= \{\neg A \text{ U } (a \wedge \mathit{weak}[G(\neg A)]) \mid a \in A^-\} \\
(A \text{ U } B)^+ &= \{(A \ \wedge \ \neg B) \text{ U } ((a \wedge \neg B) \wedge \mathit{weak}[A \text{ U } B]) \mid a \in A^+\} \cup \\
&\qquad \{(A \wedge \neg B) \text{ U } b \mid b \in B^+\} \\
(A \text{ U } B)^- &= \{(A \wedge \neg B) \text{ U } ((a \wedge \neg B) \mid a \in A^-\} \cup \\
&\qquad \{(A \wedge \neg B) \text{ U } (b \wedge (\neg \mathit{weak}[A \text{ U } B])) \mid b \in B^-\} \\
X(A)^+ &= \{X(a) \mid a \in A^+\} \\
X(A)^- &= \{X(a) \mid a \in A^-\}
\end{aligned}
$$

Table 3.10: Weakened UFC LTL Formulas describing Accepting Prefixes

The only formulas that are changed in Table 3.10 from the original formulation in Section 3.1.3 are $G(A)^+$, $F(A)^-$, one branch of $(A \text{ U } B)^+$, and one branch of $(A \text{ U } B)^-$. 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.

3.2 Preliminary Experiment

In this section, we describe an experiment that empirically evaluates and compares the effectiveness of the different requirements coverage metrics defined previously. The case study described in this section is largely adopted from our previous work published in [77]. To evaluate the effectiveness, we first generate a test suite that

provides the defined requirements coverage. The test suites were automatically generated using a model checker. A model checker can be used to find test cases by formulating a test criterion as a verification condition; negating the verification condition which is then termed as a *trap property*, and challenging the model checker to find a counter example that then constitutes a test case. A more detailed discussion of automated test generation using model checkers is presented in Appendix Section A.2. We then run the generated test suite on the system model and measure model coverage achieved by the requirements-based tests. We perform this evaluation for each of the three requirements coverage metrics defined in Section 3.1 and compare their effectiveness in terms of model coverage achieved.

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. A Flight Guidance System is a component of the overall Flight Control System (FCS) in a commercial aircraft. Chapter 4.3.1 provides a description of the system. This example consists of 293 informal requirements formalized as LTL properties as well as a formal model captured in our research notation RSML^{-e} [74]. All the properties in the FGS system are safety properties, there are no liveness properties.

3.2.1 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 three 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 antecedent coverage*. Consider the requirement

$$G(A \rightarrow B)$$

Informally, it is always the case that when A holds B will hold. A test case providing requirements antecedent coverage over such a requirement will ensure that the antecedent A becomes true at least once along the satisfying path. That is, the test case would satisfy the obligation

$$G(A \rightarrow B) \wedge F(A)$$

We implemented a transformation pass over the LTL specifications so that for requirements of the form $G(A \rightarrow B)$ we would generate trap properties requiring A to hold somewhere along the path.

Third, 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 3.1.3. Using these rules, we implemented a transformation pass over the LTL specifications to generate trap properties for both the neutral and weakened UFC notions discussed in section 3.1.4 (we used the same implementation to generate tests for requirements coverage, and requirements antecedent coverage mentioned above). However, both neutral and weakened notions of UFC result in the same test suite for this case example, since the LTL property set for the FGS system has no 'future' and 'until' temporal operators. We only generated the positive UFC set over the temporal properties since each of the properties is known to hold of the model.

Although the properties were already known to hold of the model, generating tests is still a useful exercise. For a developer, it provides a rich source of requirements-derived tests that can be used to test the behavior of the object code. For the purposes of this dissertation, it provides a straightforward way to test the fault finding capability and completeness of our metrics.

Test suite Generation: To generate the test cases we leveraged a test case generation environment that was built in a previous project [36]. This environment uses the bounded model checker in NuSMV for automated 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 the NIMBUS RSML^{-e} simulator 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 [33]. 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 all three test suites (requirements coverage, requirements antecedent coverage, and 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 achieved over the model. Definitions for the different coverage measures is available in Section 2.2.1

3.2.2 Results and Analysis

We used our tools to automatically generate and run three test suites for the FGS; one suite for requirements coverage, one for requirements antecedent coverage and

another one for requirements UFC coverage. The results from our experiment are summarized in Tables 3.11 and 3.12.

	Reqs. Cov.	**Antecedent Cov.**	**UFC Cov.**
Trap Properties	293	293	887
Test Cases	293	293	715
Time Expended	3 min	14 min	35 min

Table 3.11: Summary of the test case generation results.

Table 3.11 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 and requirements antecedent coverage test suites and can therefore be expected to provide better coverage of the model than the other two test suites. Also, the time expended in generating the UFC coverage test suite was significantly higher than the time necessary to generate the other two test suites.

We observe that our algorithm generating UFC over the syntax of the requirements generated 887 trap properties. Nevertheless, only 715 of them generated counterexamples. 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. In each of these cases the original formula was *vacuous* [7], that is, the atomic condition was not required to prove the original formula. We discuss the issue of vacuity checking further in Section 6.1.

Although it was possible to explain many of the vacuous conditions through implementation choices that satisfied stronger claims than the original properties required, the number of vacuous conditions was startling and pointed out several previously unknown weaknesses in our original property set. Rather than correcting the incor-

rectly vacuous formulas in our property set before proceeding with our experiment, we decided to use the property set "as-is" as a representative set of requirements that might be provided before a model is constructed. If test cases were manually constructed from this set of requirements, we postulate that many of these weaknesses would be found when trying to construct test cases for the vacuous conditions.

For all three coverage metrics mentioned, we did not minimize the size of the test suites generated. In previous work we found that test suite reduction while maintaining desired coverage can adversely affect fault finding [33]. However, the work in [33] was based on test suites generated using *model coverage criteria*. We plan to explore the effect of test suite reduction techniques on test suites generated using *requirements coverage criteria* in our future work.

Model Cov. Metric	**Reqs. Cov.**	**Antecedent Cov.**	**UFC Cov.**
State	37.19%	98.78%	99.12%
Transition	31.97%	89.53%	99.42%
Decision	46.42%	85.75%	83.02%
MC/DC	0.32%	23.87%	53.53%

Table 3.12: Summary of the model coverage obtained by running the requirements based tests.

In Table 3.12 we show the coverage of the formal model achieved when running the test cases providing requirements coverage, requirements antecedent coverage and requirements UFC coverage respectively. We measured four different model coverage criteria as mentioned in Section 3.2.1.

The results in Table 3.12 show that the test suite generated to provide requirements coverage (one test case per requirement) gives very low state, transition, and

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

$$G(a \rightarrow Xb)$$

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.

The requirements antecedent coverage is a stronger metric than the requirements coverage measure. It gives high state, transition and decision coverage over the model. However, the MC/DC coverage generated over the model is low. As mentioned earlier, many of the requirements in the FGS system are of the form

$$G(a \rightarrow Xb)$$

Requirements antecedent coverage will ensure that the test cases found for such properties will exercise the antecedent a (i.e., make a true). Therefore, the requirement is not trivially satisfied and we get longer test cases and, thus, better model coverage than the requirements coverage test suite.

On the other hand, the test suite generated for UFC over the syntax of the properties provides high state, transition, and decision coverage. Nevertheless, the decision coverage provided by the UFC test suite is in this experiment lower than that provided by the requirements antecedent coverage test suite. When we looked more closely at both test suites we found that for variables specified in the property (we call these *variables of interest*), both test suites had the same values. In other words, for the variables of interest, the UFC test suite is a superset of the requirements antecedent coverage test suite. However, for variables not mentioned in the properties (we call these *free variables*), the model checker has the freedom to choose any suitable value for that variable. We found that the values for these free variables differed between

the two test suites. We believe that this is the reason for the antecedent coverage test suite generating a higher decision coverage over the model than the UFC test suite; the model checking algorithm in NuSMV simply picked values for the free variables that happened to give the requirements antecedent test suite better coverage. If we could control the selection of the free variables the UFC test suite would yield the same or higher decision coverage than the requirements antecedent coverage test suite. Clearly, the test case generation method plays an important role in the types of test cases generated. We plan to explore the effect of different test case generation methods in our future work.

The UFC test suite generated low MC/DC coverage over the model, although not as low as the other two test suites. After some thought, it became clear that this is due in part 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:

$$\mathrm{G}(m\_Independent\_Mode\_Condition.result \rightarrow \mathrm{X}(Is\_This\_Side\_Active = 1))$$

Informally, it is always the case (G) 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.

$$\mathrm{G}(Macro\_name.result \rightarrow \mathrm{X}\ b)$$

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

The macro *Independent_Mode_Condition* is defined in RSML^{-e} as:

```
MACRO Independent_Mode_Condition():
   TABLE
      Is_LAPPR_Active           : T *;
      Is_VAPPR_Active           : T *;
      Is_Offside_LAPPR_Active   : T *;
      Is_Offside_VAPPR_Active   : T *;
      Is_VGA_Active             : * T;
      Is_Offside_VGA_Active     : * T;
   END TABLE
END MACRO
```

Since the structure of *Independent_Mode_Condition* 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 cover the macro.

Note that this problem is not related to the method we have presented in this dissertation; 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 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.

To get a better idea of how many additional test cases UFC coverage would necessitate when we add requirements defining macros, we considered the sample requirement property mentioned earlier:

$$\mathrm{G}(m\_Independent\_Mode\_Condition.result \rightarrow \mathrm{X}(Is\_This\_Side\_Active = 1))$$

We constructed a property defining the $Independent\_Mode\_Condition$ macro. When we performed UFC over this additional macro defining requirement we got 13 additional test cases. This result shows that adding requirements that define all the macros would make a substantial difference to the test suite size and presumably the model coverage.

Our investigation thus far illustrated that it is feasible to define a structural coverage metric over requirements and use it to measure adequacy of black box text suites. Nevertheless, there are several research problems that need to be addressed when defining an effective requirements coverage metric. We hope to address them in our future work as detailed in Chapter 7. It is also worth noting that we only investigated the effectiveness of the defined requirements coverage metrics with regard to model coverage achieved. The ultimate goal for any coverage metric is its effectiveness in revealing faults. We hope to investigate and compare the fault finding effectiveness of the different requirements coverage metrics in our future work.

3.3 Requirements Coverage Measurement Tool

Based on the definition of the requirements coverage metrics presented earlier, we have built a tool that provides the capability to measure requirements coverage achieved by a test suite. Figure 3.1 depicts how our tool works.

Our tool operates using three inputs: a test suite, a set of requirements expressed as Linear Temporal Logic (LTL) properties, and the requirements coverage metric the

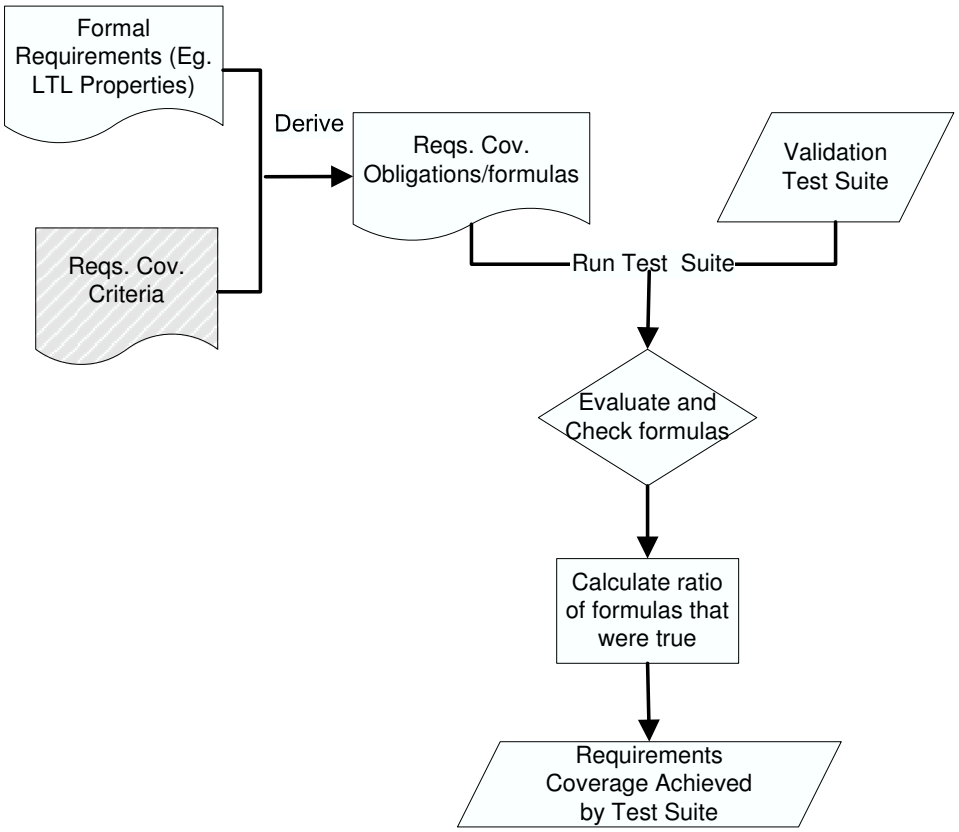

Figure 3.1: Requirements Coverage Measurement Tool

user wishes to measure. We currently support three requirements coverage metrics: Requirements Coverage, Requirements Antecedent Coverage, and Unique First Cause Coverage. The metrics were described in Section 3.1. Given these inputs, our tool outputs the requirements coverage achieved by the test suite (as per the chosen metric). Additionally, we output a file that lists the test cases that satisfy each coverage obligation and the coverage obligations satisfied by each test case. This information is expressed as an $n \times m$ array, with n test cases on the rows and m obligations marking the columns. The entries in the matrix are either 1 or 0. Thus, if a cell {i, j} in the matrix had entry 1, it indicates that the i^{th} test case *satisfied* the j^{th}

obligation. If the entry were 0, it indicates that the i^{th} test case did *not satisfy* the j^{th} obligation.

The tool operates as follows. For each requirement formalized as an LTL property, the tool first generates coverage obligations for the selected requirements coverage metric. The obligation generation is straight forward and follows the definition rules of the coverage metric. The generated obligations are LTL formulae. The LTL obligations for all the requirements are taken together as a complete set. The test suite is then run against the complete set of generated obligations to check how many of the obligations were satisfied by the test suite. To do this, the tool sequentially picks each of the test cases from the suite and runs it against each of the obligations in the complete set, and evaluates the LTL obligation using the values of variables in the test case. If the formula evaluates to a Boolean value true, then we say that the obligation is satisfied by the test case. If at least one test case in the suite satisfies the obligation, we say that the test suite satisfied the obligation. We compute the number of obligations that evaluated to true by running all the test cases against all the obligations. Finally, the tool outputs the requirements coverage achieved by the test suite as the percentage of total obligations satisfied by the test suite.

3.4 Summary

We have thus far defined coverage metrics over the structure of requirements formalized as Linear Temporal Logic (LTL) properties. These metrics are desirable because they provide *objective, model-independent* measures of adequacy for model validation activities. We defined three metrics to assess coverage over requirements:

1. Requirements Coverage

2. Requirements Antecedent Coverage

3. Unique First Cause Coverage

The metrics are ordered by the level of rigor needed in satisfying them. Requirements coverage is the weakest requiring only one test case per requirement as opposed to Unique First Cause Coverage which potentially requires several test cases per requirement (depends on the number of conditions and type of operators in the requirement). Table 3.14 shows an example LTL requirement and summarizes the obligations for the different coverage metrics. We conducted an empirical study comparing the effectiveness of the three different metrics on a close to production model of a flight guidance system. We found the UFC metric to be the most effective in terms of model coverage achieved on the flight guidance system.

We also briefly discussed how to adapt the definitions for the metrics so they can measure finite test cases. We presented 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 3.10.

Making this discussion concrete, given our example formula:

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

the UFC obligations for the original and modified rules are shown in Table 3.13. In the original formulation, the first two obligations are not satisfied by the test suite in

Table 3.8 because c never becomes true in the first test case. In the weakened formulation, however, the requirement is covered because the first test case is a potential prefix of an accepting path.

Original Fomulation:

$\{\ ((a \vee b) \wedge \neg c)\ \mathrm{U}\ (a \wedge \neg c \wedge ((a \vee b)\ \mathrm{U}\ c)),$
$((a \vee b)\ \wedge \neg c)\ \mathrm{U}\ (b \wedge \neg c \wedge ((a \vee b)\ \mathrm{U}\ c)),$
$((a \vee b) \wedge \neg c)\ \mathrm{U}\ c\ \}$

Weakened Formulation:

$\{\ ((a \vee b) \wedge \neg c)\ \mathrm{U}\ (a \wedge \neg c \wedge ((a \vee b)\ \mathrm{W}\ c)),$
$((a \vee b)\ \wedge \neg c)\ \mathrm{U}\ (b \wedge \neg c \wedge ((a \vee b)\ \mathrm{W}\ c)),$
$((a \vee b) \wedge \neg c)\ \mathrm{U}\ c\ \}$

Table 3.13: UFC obligations for $((a \vee b)\ \mathrm{U}\ c)$

Finally, we presented the requirements coverage measurement tool that provides the capability to measure requirements coverage achieved by a test suite. The tool takes as inputs the test suite, the requirements coverage metric we want to measure and the formal requirements as LTL properties. We currently support the above three requirements coverage metrics and both the neural and weakened formulations.

Formal LTL Requirement: $G(Is\_AP\_Engaged \rightarrow Onside\_FD\_On)$

Requirements Coverage

1. $G(Is\_AP\_Engaged \rightarrow Onside\_FD\_On)$

Requirements Antecedent Coverage

1. $G(Is\_AP\_Engaged \rightarrow Onside\_FD\_On) \ \wedge \ F(Is\_AP\_Engaged)$

Requirements UFC Coverage

1. $(Is\_AP\_Engaged \ \rightarrow \ Onside\_FD\_On) \ U \ ((\neg Is\_AP\_Engaged \ \wedge \ \neg Onside\_FD\_On) \wedge \ G(Is\_AP\_Engaged \rightarrow Onside\_FD\_On))$

2. $(Is\_AP\_Engaged \ \rightarrow \ Onside\_FD\_On) \ U \ ((Is\_AP\_Engaged \ \wedge \ Onside\_FD\_On) \wedge \ G(Is\_AP\_Engaged \rightarrow Onside\_FD\_On))$

Table 3.14: Example LTL property and obligations for different requirements coverage metrics

Chapter 4

Automated Requirements-Based Test Case Generation

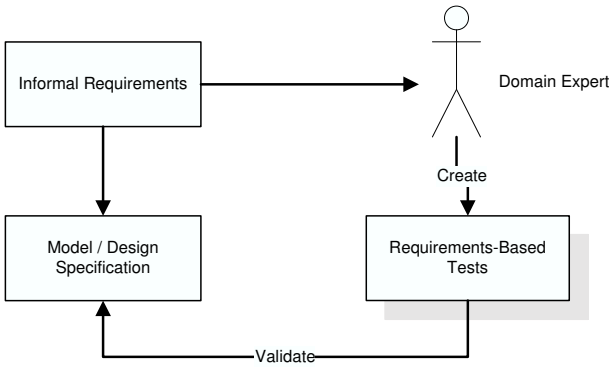

Figure 4.1: Traditional Validation Approach

Traditionally, software validation has been largely a manual endeavor wherein developers *manually* create requirements-based tests and inspect the system design, or model in model-based software development, to ensure it satisfies the requirements. Figure 4.1 illustrates the traditional validation approach. In the critical systems domain, the validation and verification (V&V) phase can be very costly and consumes a majority of the development resources. We attempt to reduce the validation effort by proposing an approach that *automates* the generation of requirements-based tests. The proposed approach and empirical study in this chapter is largely adopted from our previously published work in [61].

Our approach uses a formalized set of requirements, as illustrated in Figure 4.2,

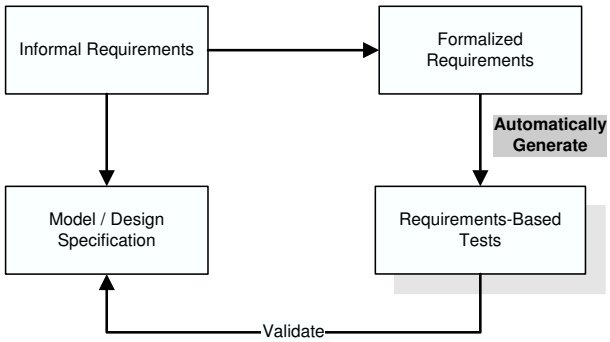

Figure 4.2: Proposed Model Validation Approach

as the basis for automated requirements-based test case generation. We defined a collection of coverage metrics over the structure of formal requirements in Section 3.1. Using the formal requirements and the defined requirements coverage metrics, we automatically generate requirements-based tests through an abstract model—we call it the *Requirements Model*—to provide coverage over the requirements (as opposed to over the model). Similar to previous research efforts [64, 66, 27], we automatically generate tests from a formal model using model checkers.

Previously, as described in Section 3.2, we developed an approach and tool support to automatically generate test cases to provide coverage of the requirements (as opposed to the model) [77]. This approach, however, uses the system design or Model Under Test (MUT) as the basis for the test case generation. In short, our tool uses the system model to find execution traces that demonstrate that the requirements are met (up to some predefined coverage of the requirements, for example, Unique First Cause coverage). Although this approach will help us find test cases, the tests are derived from the system model itself and we are explicitly searching for execution traces of the system model that satisfy the requirements; if there is at least one execution trace that would satisfy the requirement we will find that as a test case. Therefore, our earlier approach is useful to *illustrate how* a system model can satisfy

its requirements, but it is not suitable to investigate *whether or not* the system model satisfies its requirements. To address the latter issue it is desirable to somehow generate test cases directly from the requirements without referring to the behavior of the MUT.

To achieve this goal, we alter our previous approach so that it uses an abstract model derived only from the requirements without any information about the behavior of the MUT—we call this abstract model the *Requirements Model*. In general, when testing to check whether an artifact satisfies a set of requirements, it is highly undesirable to derive the tests with guidance from the artifact under test. The *requirements model* must therefore contain only the behaviors *required* from the MUT, and not be constrained to the behaviors actually *provided* by the MUT. Given formalized requirements, constraints on the environment of the system, and information about the types of the inputs and outputs of the MUT, we can create an abstract model—independent from the MUT—that captures only the required behavior. In our work we create the requirements model by encoding the requirements and environmental assumptions as invariants. By using this requirements model as a basis for test case generation we can generate truly black-box requirements-based tests for MUT validation.

We assess the effectiveness of the proposed approach for validation on three realistic models from the avionics domain: the Mode Logic for a Flight Guidance System (FGS), and two models related to the display window manager for a new airliner (DWM_1, and DWM_2). For each of these models we automatically generate requirements-based tests using the requirements model to provide Unique First Cause (UFC) coverage over the formal requirements (UFC definition is presented in Section 3.1). We then run the generated test suite on the MUT and measure the *model coverage* achieved. In particular, we measure MC/DC achieved over the MUT.

From our experiment we found that the requirements-based tests did extremely well on the DWM_1 and DWM_2 systems (both production models) achieving more than 95% and 92% MC/DC coverage over the models. On the other hand, the requirements based tests for the FGS (a large case example developed for research) performed poorly providing only 41% coverage of the FGS model. We hypothesize that the poor results on the FGS were due to the inadequacy in the FGS requirements set; the requirements sets for the DWM_1 and DWM_2 systems, on the other hand, were developed for production and were extensive, well validated, and well defined. These experiences indicate that our approach can be effective in the validation testing of models in model-based development. In addition, the capability of measuring coverage of the requirements as well as the model enables us to assess the adequacy of a set of requirements; if we cover the requirements but not the model—as in the case of the FGS mentioned above—it is an indication that the we have an incomplete set of requirements.

In the remainder of this chapter, we describe our proposed test case generation technique in detail in Sections 4.1 and 4.2. We describe the experiment conducted to evaluate our approach in Section 4.3. Results obtained and their analysis is presented in Section 4.4. Section 4.5 discusses the implications of our experimental results.

4.1 Requirements-Based Test Case Generation Using Model Checkers

The technique for automatically generating tests from formal models using model checkers is described in Section A.2. In this section, we describe how this technique can be tailored to automatically generate test cases providing requirements coverage. The requirements coverage metric used in our experiment is the *Unique First Cause*

(UFC) coverage defined in [77] and described in Section 3.1. We evaluated the feasibility of the proposed approach using only the UFC metric for requirements coverage. We chose the UFC metric for our investigation since it is the most rigorous amongst the defined metrics. In the future, we plan to evaluate the effectiveness using other requirements coverage metrics as well.

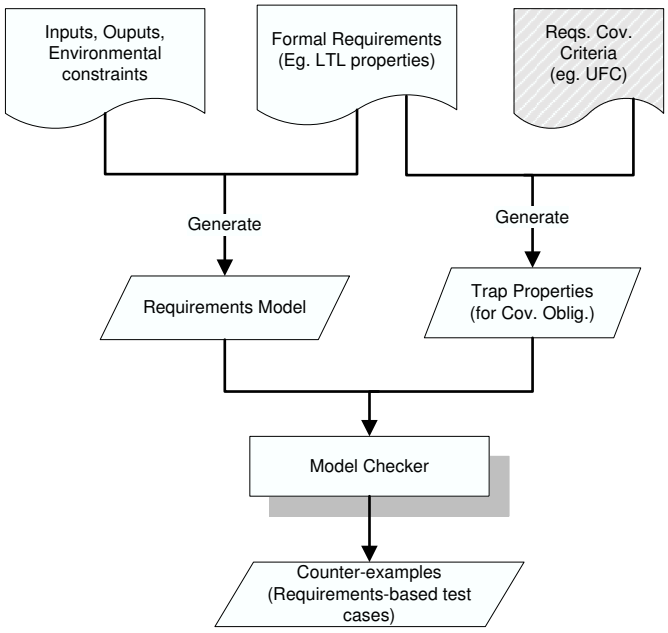

Figure 4.3: Automated Requirements-Based Test Case Generation

Earlier we briefly described UFC over paths. Using this definition we can derive UFC obligations that show that a particular atomic condition affects the outcome of the property. Given these obligations and a formal model of the software system, we can now challenge the model checker to find an execution path that would satisfy one of these obligations by asserting that there is no such path (i.e., negating the obligation). As mentioned in Section A.2, we call such a formula a trap property. 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 UFC obligation of interest over the model. By repeating this process for all UFC obligations within the set derived from a property, we can derive UFC coverage of the property over the model. By performing this process on all requirements properties, we can derive a test suite that provides UFC coverage of the set of requirements. This process is illustrated in Figure 4.3.

When the model checker does not return a counterexample (or test case) for a trap property (in our case for an UFC obligation) it means that a test case for that particular test obligation does not exist. In the case of UFC obligations it implies that the atomic condition that the obligation was designed to test *does not* uniquely affect the outcome of the property. In each of these cases the original requirements property is *vacuous* [7], that is, the atomic condition is not required to prove the original property.

For reasons mentioned earlier in the section, we need to create a Requirements Model different from the MUT for requirements-based test case generation.

4.2 Requirements Model for Test Case Generation

The requirements model is created using the following information:

- requirements specified as invariants
- inputs, and the outputs of the MUT
- input constraints or environmental assumptions (if any)

The formalized requirements and environmental assumptions are specified as invariants in the requirements model. These invariants restrict the state space of the requirements model so that we only allow behaviors defined by the requirements. We built the requirements model in this fashion since tests derived for model validation

should be based solely on the high-level requirements and the environmental assumptions, and should not be influenced by the structure and behavior of the MUT. In addition, the names of the inputs and outputs of the MUT are needed to construct concrete test cases that can be executed on the MUT.

We hypothesize that with a well defined set of requirements and environmental constraints, requirements-based tests generated from the requirements model to provide UFC coverage of the requirements will provide high MC/DC coverage of the model under test and, thus, be highly beneficial in the validation testing process. We empirically evaluate this hypothesis in Section 4.4 using three realistic examples from the avionics domain.

We developed a tool that allows the requirements model to be built in an automated fashion. The tool takes as inputs a formal set of requirements, environmental constraints, and the MUT. Requirements need to be formalized in the LTL notation and the environmental constraints specified as invariants. The tool starts with the MUT and strips out everything but the declaration for inputs and outputs of the MUT. To this stripped model (only containing the declarations for the inputs and outputs of the MUT), the tool automatically adds the formal LTL requirements and environmental constraints as invariants. The resulting model is the requirements model. Figure 4.4 illustrates the approach used in the tool to create the Requirements Model.

The tool currently supports only requirements expressed as safety properties; requirements expressed as liveness properties are not yet supported. Informally, a safety property stipulates that some "bad thing" will never happen during execution of a program and a liveness property stipulates that some "good thing" will eventually happen. In our work we have not found this to be a limitation since *all* requirements expressed over our case-examples can be expressed as safety properties. Converting

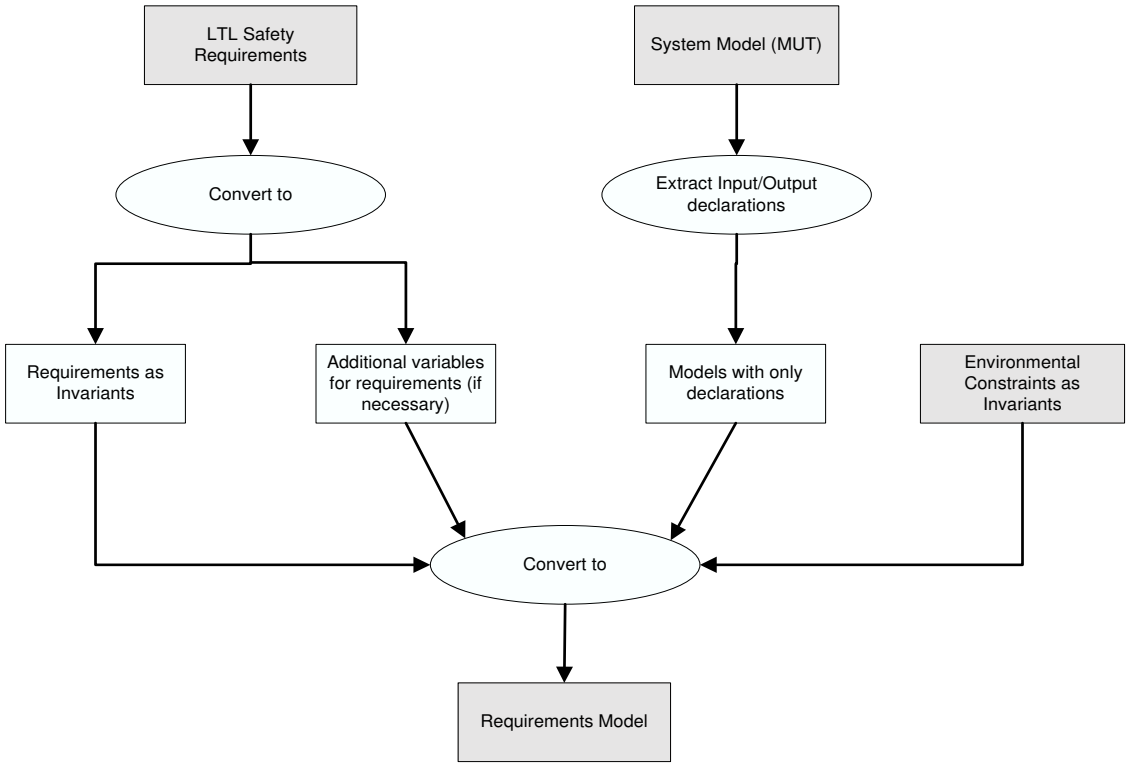

Figure 4.4: Approach used in Tool that Automatically Creates the Requirements Model

safety properties into invariants in the requirements model is not a trivial task. The tool supports the SMV notation for the models [55]. Note that in the SMV notation, the declaration that allows us to specify a set of invariant states ("INVAR") only allows boolean expressions in its syntax. Thus for requirements defined using temporal operators, SMV will not allow them to be used directly in the "INVAR" declaration. For requirements containing the next state temporal operator, this issue can be resolved easily by defining additional variables in the requirements model. To see this, consider the following example of a naïve informal requirement:

```
"If Switch is pressed, then in the next step the Light will turn
on."
```

formalized in LTL as:

```
LTLSPEC G((Switch = pressed) -> X (Light = on))
```

The formalized requirement has the temporal next state operator X in the expression within the global operator G. To express the above requirement as an invariant in the requirements model, we will declare a new variable, req_1 to help us in the invariant definition.

```
ASSIGN
    init(req_1) := TRUE;
    next(req_1) := (Switch = pressed) -> next(Light = on);


INVAR req_1
```

As seen in the SMV assignment above, we specify the requirement as an invariant using this new variable. If the requirement does not contain any temporal operators, we do not need this additional variable. We can simply put the requirements expression in the invariant declaration (INVAR). This method of specifying invariants will

not work for requirements defined using "Future" and "Until" LTL operators. Our tool does not currently support such requirements. We are investigating this issue and plan to resolve it in our future work. (As mentioned above, this restriction has not been a problem in practice since all requirements we have encountered in our case examples could be formalized as safety properties.)

For the example requirement above, test cases through the requirements model that provide UFC coverage over the requirement may have the following form:

```
------------------------
    Test Case 1
------------------------
Step 1
    Switch = not pressed
    req_1 = true
    Light = off


------------------------
    Test Case 2
------------------------
Step 1
    Switch = not pressed
    req_1 = true
    Light = off
Step 2
    Switch = pressed
    req_1 = true
    Light = off
```

```
Step 3
    Switch = pressed
    req_1 = true
    Light = on
```

The above two test cases can now be run through the MUT using the values of inputs from the test cases. If *Switch* was the only input to the MUT, and *Light* was an output; running the test cases through the MUT would constitute looking up the value of *Switch* at each step in the test case and calculating the output *Light* through the MUT.

4.3 Experiment

In this experiment we were interested in determining (1) the feasibility of generating requirements-based tests from a requirements model, and (2) the effectiveness of these test sets in validating the system model or MUT. We evaluated the effectiveness of the generated test cases using three realistic examples from the avionics domain - the FGS, and two models related to the Display Window Manager system (DWM_1, and DWM_2). We provide a description of these systems in Section 4.3.1. The requirements for these systems were formalized as LTL properties. We generated test cases to provide UFC coverage over the LTL properties. In this experiment we assessed the effectiveness of the test sets in terms of coverage achieved over the MUT since this is a major concern in our application domain. In the future, we plan on evaluating the quality of these test sets in terms of their fault finding capability on the model.

4.3.1 Case Examples

In the experiment in this chapter we used three close to production or production systems, namely: two models related to the Display Window Manager System, and a prototype model of a Flight Guidance System. All of these systems were provided to us by Rockwell Collins Inc through our long standing collaboration with Dr. Michael Whalen. We provide descriptions for these systems in this section. Note that in the description of the FGS, we refer to two systems modeling the mode logic of the FGS (vertical and lateral mode logic) in addition to the prototype model used in this chapter. These two systems are used in our empirical investigation described in Chapter 5; they are referred to here to provide a complete description of the FGS and to avoid repetition.

Display Window Manager Models (DWM_1 and DWM_2):

The Display Window Manager models, DWM_1, and DWM_2, represent 2 of the 5 major subsystems of the Display Window Manager (DWM) of an air transport-level commercial displays system. Both the DWM_1 and DWM_2 subsystems were modeled using the Simulink notation from Mathworks Inc. The DWM acts as a 'switchboard' for the system and has several responsibilities related to routing information to the displays and managing the location of two cursors that can be used to control applications by the pilot and copilot. The DWM must update which applications are being displayed in response to user selections of display applications, and must handle reversion in case of hardware or application failures, deciding which information is most critical and moving this information to the remaining display(s). It also must manage the cursor, ensuring that the cursor does not appear on a display that contains an application that does not support the cursor. In the event of reversion, the DWM must ensure that the cursor is not tasked to a dead display.

The DWM_1 system consists of 43 informal requirements formalized as LTL properties. The formal model of the system was built in the Simulink notation from Mathworks, Inc [51]. The DWM_2 system consists of 85 informal requirements formalized as LTL properties. The formal model of the system was built in the Simulink notation.

Flight Guidance System:

A Flight Guidance System is a component of the overall Flight Control System (FCS) in a commercial aircraft. It compares the measured state of an aircraft (position, speed, and altitude) to the desired state and generates pitch and roll-guidance commands to minimize the difference between the measured and desired state. The FGS consists of the mode logic, which determines which lateral and vertical modes of operation are active and armed at any given time, and the flight control laws that accept information about the aircraft's current and desired state and compute the pitch and roll guidance commands.

This system consists of 293 informal requirements formalized as LTL properties. We use three FGS models in our empirical investigations that focus on the mode logic of the FGS. The first model, ToyFGS, is a prototype model described in [54] and the formal model is captured in our research notation RSML^{-e} [74]. The ToyFGS, although named with "Toy", is a close to production model. To avoid confusion, in the rest of this dissertation, we will refer to the ToyFGS model as the FGS model. The other two models, Vertmax_Batch and Latctl_Batch models, describe the vertical and lateral mode logic for the flight guidance system and are part of the Rockwell Collins FCS 5000 flight guidance system family, described in [52]. The formal models for these two systems were captured in the Simulink notation from Mathworks, Inc.

4.3.2 Setup

The experiment constituted the following steps:

Create the Requirements Model: The requirements model as mentioned before was built using the formalized set of requirements, names of inputs and outputs of the MUT, and environmental assumptions for the system. We described the tool to build the requirements model in an automated fashion in Section 4.2. The modeling notation that we use in our tool is the SMV language.

Generate Requirements Obligations: We started with the set of requirements formalized as LTL properties. We generated obligations (as LTL specifications) to provide *requirements UFC coverage* over the syntax (or structure) of the LTL properties. The rules and tool to auto-generate UFC obligations was described in Section 3.

Generate Requirements-Based Test Cases: We used the bounded model checker in NuSMV for automated test case generation. We generated test cases from the requirements model to provide UFC coverage over the properties. We discussed this approach in Section 4.1 previously. We ran the generated test suite on the MUT to measure the resulting model coverage achieved.

Measure Model Coverage Achieved: In this experiment we assessed the effectiveness of the test sets in terms of coverage achieved over the MUT since this is a major concern in our application domain. (In the future, we plan on evaluating the quality of these test sets in terms of their fault finding capability on the MUT.) To measure coverage achieved by the requirements-based test suites over the RSML^{-e} model of the FGS, we leveraged tools built in a previous project [33] that allowed us to measure different kinds of coverage of

	# Obligations	# Tests	Time Expended	MC/DC Ach.
FGS	887	835	47 mins	41.7%
DWM_1	129	128	$<$ 1 min	95.3%
DWM_2	335	325	$<$ 2 mins	92.6%

Table 4.1: Summary of Requirements-Based Tests Generated and Coverage Achieved

the FGS model expressed in RSML^{-e}. To measure coverage over the DWM_1 and DWM_2 models in Simulink, we used the capabilities in our translation infrastructure [75]. The infrastructure allows us to translate the Simulink model into a synchronous language, Lustre [30]. We then measure coverage over the translated Lustre model. Using these measurement tools, we ran the test suite providing requirements UFC coverage and recorded coverage achieved over the MUT. In particular, we measured MC/DC achieved over the MUT. We chose to measure this coverage since current practices and standards [70] in the avionics domain require test suites to achieve MC/DC over the software.

4.4 Experiment Results and Analysis

Table 4.1 shows the coverage achieved by the requirements-based tests over the MUT for the three systems. The requirements based tests did poorly on the FGS covering a mere 41% of the model. On the other hand, the tests did well on the DWM_1 and DWM_2 systems covering more than 95% and 92% of the MUT respectively. Our findings and analysis for the systems are summarized in the remainder of this section.

4.4.1 FGS

As seen from Table 4.1, we generated 887 requirements UFC obligations for the FGS requirements, of which 835 resulted in test cases. The time expended in test generation was 47 minutes using the NuSMV bounded model checker. For the remaining 52 obligations that did not result in test cases, 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 and, thus, no test case demonstrating this independent effect exists. The inability to find a test case may be an indication of a poorly written requirement or an inconsistent set of requirements. In this report, we did not attempt to correct these properties, we decided to use the property set "as-is" as a representative set of requirements that might be provided before a model is constructed.

Table 4.1 shows that the test suite generated to provide UFC over the FGS requirements provides only 41% MC/DC coverage over the MUT. To explain the poor coverage, we took a closer look at the requirements set for the FGS and found that the poor performance of the generated test suite was due in part 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:

$$\mathrm{G}(m\_Independent\_Mode\_Condition.result \rightarrow \mathrm{X}(Is\_This\_Side\_Active = 1))$$

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.

$$\mathrm{G}(Macro\_name.result \rightarrow \mathrm{X}\ b)$$

The definition of the macro resides in the MUT and is missing from the property set. Therefore, when we perform UFC over this property structure, we do not perform

UFC over the—potentially very complex—condition making up the definition of the macro.

The macro *Independent_Mode_Condition* is defined in RSML^{-e} as:

```
MACRO Independent_Mode_Condition():
 TABLE
   Is_LAPPR_Active           : T *;
   Is_VAPPR_Active           : T *;
   Is_Offside_LAPPR_Active   : T *;
   Is_Offside_VAPPR_Active   : T *;
   Is_VGA_Active             : * T;
   Is_Offside_VGA_Active     : * T;
 END TABLE
END MACRO
```

(The table in the macro definition is interpreted as a Boolean expression in disjunctive normal form; each column in the table represents one disjunction; a * indicates that in this disjunction the condition on that row is a don't care.) Since the structure of *Independent_Mode_Condition* is not captured in the required property, the test cases generated to cover the property will not be required to exercise the conditions making up the definition of the macro. We will thus most likely only cover one of the UFC cases needed to adequately cover 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 formalization of the FGS requirements as LTL properties. Properties should not be stated using internal variables, functions, or macros of the MUT; doing so leads to a level of circular reasoning (using concepts defined in the model to state properties of the model). If a property must be stated using an internal variable (or macro) then additional requirements (properties)

are needed to define the behavior of the internal variable in terms of inputs to the system. For the FGS, 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 MUT.

Thus, for the FGS, our approach helped identify inadequacies in the requirements set by measuring coverage achieved on the MUT with the generated requirements-based tests. Rockwell Collins Inc. was aware of the problems related to defining requirements using internal variables from the model, learned from the FGS research model, and rectified this problem in later modeling efforts. The DWM models presented in the next section are two such models with well defined requirements.

4.4.2 DWM_1 and DWM_2

In contrast to the FGS, in the DWM models all the requirements were defined completely in terms of inputs and outputs of the system. Internal variables—if any—used when describing requirements were defined with additional requirements. The problems seen with the requirements of the FGS were not present in the DWM examples. As seen in Table 4.1, our test case generation approach was feasible on both the DWM_1 and DWM_2 systems. For the DWM_1 system, we generated 128 requirements-based tests from 43 formalized requirements in less than one minute. The generated requirements-based tests covered more than 95% of the MUT. On the DWM_2 system, we generated 325 test cases from 85 requirements in less than two minutes. The generated tests covered more than 92% of the MUT. Note that 10 of the 335 UFC obligations on the DWM_2 system did not generate test cases. This implies that the atomic conditions that these obligations were designed to test were vacuous in the requirement. It is evident from these results that in both the systems,

"If a is true then in the next step b will be true"	(1)
$G(a \rightarrow Xb)$	(2)
$(a \rightarrow Xb)\ U\ ((a\ \&\ Xb)\ \&\ G(a \rightarrow Xb))$	(3)

Table 4.2: (1) Example high-level requirement (2) LTL property for the requirement (3) UFC obligation for atomic condition b

the requirements-based tests cover most of the behavior in the MUT and therefore have the potential to be effective in model validation.

In our experiments we observed that for some of the generated requirements-based tests the outputs predicted by the requirements model differed from the outputs generated when the tests were executed on the MUT. This occurs because the MUT may define constraints not in the requirements model, constraints that cause the test cases to lead to different outputs. To illustrate, consider the naïve example of a requirement and one of its UFC obligation in Table 4.2. Let us suppose a is an input and b an output of the example system. The UFC obligation states that $(a \rightarrow Xb)$ is true until you reach a state where a is true and in the next state b is true, and the requirement continues to hold thereafter. This obligation would ensure that b is necessary for the satisfaction of the requirement.

For illustration purposes, let us suppose the MUT is built in such a way that it imposes an additional constraint: `"Output b is always true"`; this model would still satisfy the requirement. Table 4.3 shows a requirements-based test case predicting a certain behavior through the requirements model but when executed through the MUT we get a different (but still correct) behavior. The test case results in different values for output b between the requirements model and the MUT because of the the additional constraint imposed by the MUT.

Requirements Model

Step	1	2	3	4
a	0	0	1	0
b	0	0	0	1

MUT

Step	1	2	3	4
a	0	0	1	0
b	1	1	1	1

Table 4.3: Sample Test Execution through Requirements Model and MUT

Observing such differences may help developers in validating the additional constraint imposed by the MUT. Note again that differences in test results predicted by the requirements model and the actual results from the MUT do not imply that the MUT is incorrect. The MUT may be correct but simply more restrictive. Our approach does not address this oracle problem at this time and we currently rely on the developers to make a decision of whether or not the results are acceptable. (A more extensive discussion on this issue is provided in Section 4.5). In our experience, additional constraints in the MUT are usually correct and needed, and they are missing from the requirements set. Thus in addition to validation this exercise may help developers in identifying these missing requirements.

The requirements-based tests generated using our approach provides very high coverage over the MUT for the DWM_1 and DWM_2 systems owing to their well defined set of requirements. Nevertheless we did not get 100% coverage over these systems. There may be several factors contributing to this result: (1) there may be missing requirements, (2) the model is violating some of the requirements, and (3) there may be a mismatch between our definition of UFC coverage in the requirements

domain and the MC/DC coverage as measured in the model domain. Presently, we have been unable to determine which of these was the contributing factor on the two systems, but we plan to investigate it in the future.

4.5 Discussion

In our experiment, we found that our proposed approach for automatically generating validation tests is feasible but the effectiveness of the generated tests is (not unexpectedly) subject to the quality of the requirements. For an incomplete set of requirements—such as the one provided for the FGS—the requirements-based tests provide low coverage over the MUT. Note that this is a problem with the requirements and not the proposed approach. Nevertheless, for a mature and extensive set of requirements—such as those provided for the DWM models—the generated requirements-based tests provide high coverage over the MUT. Additionally, our approach has the potential to help identify missing requirements, like in the case of the FGS. When requirements-based tests providing coverage over the requirements provide poor coverage over the model it is an indication that we have an incomplete set of requirements.

When validating and analyzing models using the generated requirements-based tests, we encourage developers to carefully consider the following two issues.

First, does every requirements coverage obligation result in a test case from the requirements model? If it does not it is an indication that the requirement (property) from which the obligation is derived is poorly written. For instance, if requirements UFC coverage is used it means that the condition that the obligation is designed to test *does not* uniquely affect the outcome of the property and is thus not required to demonstrate that the property holds. Investigating this issue further would help in getting better quality requirements.

Second, do the predicted outputs match the actual outputs? Compare the predicted outputs of the requirements-based tests from the requirements model with the outputs as they are executed on the MUT. As seen in our experiment, for all three systems, the predicted outputs frequently differ from the output actually produced by the MUT. We found that this occurred because of the additional constraints that were in the MUT but not in the requirements model. Investigating this discrepancy in outputs will help validate these additional constraints.

The oracle problem, that is, deciding whether the generated requirements-based tests pass/fail on the MUT is not handled in this dissertation. As discussed, this decision cannot be made by simply comparing the outputs from the MUT with the ones predicted by the requirements model. We are currently investigating techniques that will automate the oracle problem in the future. One simple solution would be to discard the expected outputs generated in the requirements-based test case generation, run the tests through the MUT and collect the execution traces, and then use the requirements and the requirements coverage obligations as monitors over the traces to determine if a violation has occurred. This would be a simple solution drawing on readily available techniques from the run-time verification community.

Finally, it is worth noting that the requirements coverage metric used plays a key role in the effectiveness of the generated requirements-based tests. For instance, if we use decision coverage of the requirements, it would require a single test case that demonstrates that the requirement is satisfied (a negative test case that demonstrates that the requirement is not met would presumably not exist). Typically, a single test case per requirement is too weak of a coverage since it is possible to derive many rather useless test cases. If we again consider our sample requirement and formalization from Table 3.2. We can, for example, satisfy the decision coverage metric by creating a test case that leaves the autopilot disengaged throughout the test

and disregards the behavior of the flight director. Although this test case technically satisfies the requirements, it does not shed much light on the correctness of the MUT. It is therefore important to choose a rigorous requirements coverage metric, like the UFC coverage, for requirements-based test case generation.

Chapter 5

Requirements Coverage as an Adequacy Measure for Conformance Testing

In model-based software development, the traditional testing process is split into two distinct activities: one activity that tests the model to *validate* that it accurately captures the customers' high-level requirements, and another testing activity that *verifies* whether the code generated (manually or automatically) from the model is behaviorally equivalent to (or conforms to) the model. In this chapter, we focus on the second testing activity—verification through conformance testing. There are currently several tools, such as model checkers, that provide the capability to automatically generate conformance tests [65, 27] from formal models. In this chapter, we examine the effectiveness of metrics used in measuring the adequacy of the generated conformance tests. In particular, we examine whether requirements coverage metrics can aid in measuring adequacy of conformance test suites.

For critical avionics software, DO-178B necessitates test cases used in verification to achieve requirements coverage in addition to structural coverage over the code. However, previously, there was no direct and objective measure of requirements coverage, and adequacy of tests was instead inferred by examining structural coverage achieved over the model. The Modified Condition and Decision Coverage (MC/DC) used when testing highly critical software [70] in the avionics industry has been a natural choice to measure structural coverage for the most critical models. As seen

in Chapter 3, however, we have defined coverage metrics that provide *direct and objective* measures of how well a test suite exercises a set of high-level formal software requirements. In Section 3.2, we examined using requirements coverage metrics, in particular the Unique First Cause (UFC) coverage metric, to measure adequacy of tests used in validation (or black-box testing) and found them to be useful. To save time and effort, we would like to re-use validation tests providing requirements coverage for verification of code through conformance testing as well. This chapter examines the suitability of using tests providing requirements UFC coverage for conformance testing as opposed to tests providing MC/DC over the model. The empirical investigation discussed in this chapter has largely been adopted from our previous work published in [63].

We believe requirements coverage will be useful as an adequacy measure for conformance testing for several reasons. First, measuring structural coverage over the requirements gives a direct assessment of how well the conformance tests exercise the required behavior of the system. Second, if a model is missing functionality, measuring structural coverage over the model will not expose such defects of omission. Third, obligations for requirements coverage describe satisfying scenarios (paths) in the model as opposed to satisfying states defined by common model coverage obligations (such as MC/DC). We hypothesize that the coverage obligations that define satisfying paths will necessitate longer and more effective test cases than those defining satisfying states in the model. Finally, we found in [62] that structural coverage metrics over the model, in particular MC/DC, are sensitive to the structure of the model used in coverage measurement. Therefore, these metrics can be easily rendered inefficient by (purposely or inadvertently) restructuring the model to make it easier to achieve the desired coverage.

For these reasons, we believe that requirements coverage will serve as a stronger adequacy measure than model coverage in measuring adequacy of conformance test suites. More specifically, we investigate the following hypothesis:

Hypothesis 1 (H_1): *Conformance tests providing requirements UFC coverage are more effective at fault finding than conformance tests providing MC/DC over the model.*

We evaluated this hypothesis on four industrial examples from the civil avionics domain. The requirements for these systems are formalized as Linear Temporal Logic (LTL) [22] properties. The systems were modeled in the Simulink notation [51]. Using the Simulink models, we created implementations using our translation infrastructure [75] in the synchronous dataflow language Lustre [30]. We used the implementations in Lustre as the basis for the generation of large sets of mutants by randomly seeding faults. We generate numerous test suites to provide 100% achievable UFC coverage over the LTL properties (the formal requirements), and numerous test suites to provide 100% achievable MC/DC over the model. We assessed the effectiveness of the different test suites by measuring their fault finding capability, i.e., running them over the sets of mutants and measuring the number of faults detected.

In our experiment we found that Hypothesis 1 was rejected on three of the four examples at the 5% statistical significance level. This result was somewhat disappointing since we believed that the requirements coverage would be effective as a conformance testing measure. The astute reader might point out that the result might not be surprising since the effectiveness of the requirements-based tests providing UFC coverage heavily depends on the 'goodness' of the requirements set; in other words, a poor set of requirements leads to poor tests. In this case, however, we worked with case examples with very good sets of requirements and we had expected better results. Nevertheless, we found that the tests providing requirements UFC

coverage found several faults that remained undetected by tests providing MC/DC over the model. We thus formed a second hypothesis stating that complementing model coverage with requirements coverage will prove more effective as an adequacy measure than solely using model coverage for conformance testing. To investigate this, we formulated and tested the following hypothesis:

Hypothesis 2 (H_2): *Conformance tests providing requirements UFC coverage in addition to MC/DC over the model are more effective at fault finding than conformance tests providing only MC/DC over the model.*

In our second set of experiments, the combined test suites were significantly more effective than MC/DC test suites on three of the four case examples (at the 5% statistical significance level). For these examples, UFC suites found several faults not revealed by the MC/DC suites making the combination of UFC and MC/DC more effective than MC/DC alone. The relative improvement in fault finding over the MC/DC suites was in the range of $4.3\% - 10.8\%$ on these examples. We strongly believe that for the case example that did not support Hypothesis 2, the MC/DC suite found all possible faults, making improvement with the combined suites impossible. Based on our results, we believe that existing adequacy measures for conformance testing based solely on structural coverage over the model (such as MC/DC) can be strengthened by combining them with requirements coverage metrics such as UFC. It is worth noting that Briand et.al. found similar results in their study [14], though in the context of state-based testing for complex component models in object-oriented software. Combining a state-based testing technique for classes or class clusters modeled with statecharts [31], with a black-box testing technique, category partition testing, proved significantly more effective in fault detection. We recommend, based on our results, that *future measures of conformance testing adequacy consider both requirements and model coverage either by combining existing metrics, such as MC/DC*

and UFC, or by defining new metrics that account for behaviors in the requirements in addition to those in the model.

Achieving structural coverage over requirements usually necessitates longer test cases that reveal different faults than those necessary to achieve structural coverage over the model. Nevertheless, it is important to keep in mind that any approach, including ours, that rely on structural coverage metrics will be highly sensitive to the structure of the artifact being covered. For instance, in this experiment we found that the UFC metric was surprisingly sensitive to the structure of the requirements, and one has to ensure that the requirements structure does not hide the complexity of conditions for the metric to be effective. Section 5.3.1 illustrates this issue in more detail.

The remainder of the chapter is organized as follows. Section 5.1 introduces our experimental setup. Results and statistical analysis are presented in Section 5.2. Finally in Sections 5.3 and 5.4, we analyze and discuss the implications of our results, and point to future directions.

5.1 Experiment

We use four industrial systems in our experiment: two models from a display window manager for an air-transport class aircraft (DWM_1, DWM_2), and two models representing flight guidance mode logic for business and regional jet class aircrafts (Vertmax_Batch and Latctl_Batch). All four systems were viewed to have good sets of requirements. Description for these systems were previously provided in Section 4.3.1. We conducted the experiments for each case example using the steps outlined below (elaborated in later sections):

1. **Generate and Reduce Test Suites to provide UFC Coverage:** We generated a test suite to provide UFC coverage over the formalized LTL requirements. This test suite was naïvely generated, one test case for every UFC obligation, and thus highly redundant. We reduced the test suite randomly while maintaining UFC coverage over the requirements. We generated three such randomly reduced test suites.

2. **Generate and Reduce Test Suites to provide MC/DC over the model:** We naïvely generated a test suite to provide MC/DC over the model. We then randomly reduced the test suite to maintain MC/DC over the model. We generated three such reduced test suites.

3. **Combined test suites that provide MC/DC + requirements UFC:** Among the reduced MC/DC suites from the previous step, we select the most effective MC/DC test suite based on their fault finding ability. We merge this test suite with each of the reduced UFC test suites from the first step. The combined suites thus provide both MC/DC over the model and UFC coverage over the requirements.

4. **Generate Mutants:** We randomly seeded faults in the correct implementation and generated three sets of 200 mutants using the method outlined in Section 5.1.2.

5. **Assess and compare fault finding:** We run each of the test suites from steps 1, 2 and 3 (that provide requirements UFC coverage, MC/DC over the model, and MC/DC + requirements UFC coverage respectively) against each set of mutants and the model. Note that the model serves as the oracle implementation in conformance testing. We say that a mutant is killed (or detected) by a test suite if any of the test cases in the suite result in different output values

between the model and the mutant. We recorded the number of mutants killed by each test suite and computed the fault finding ability as the percentage of mutants killed to the total number of mutants.

5.1.1 Test Suite Generation and Reduction

We generated test suites to provide UFC coverage over formal LTL requirements and to provide MC/DC over the model. The approach to generate and reduce the test suites for the two different coverage measures is detailed below. Additionally, we merge the reduced test suites for the two coverage measures to create combined test suites that provide UFC coverage over the requirements in addition to MC/DC over the model.

UFC Coverage over Requirements:

The requirements coverage metric used in this report is the *Unique First Cause* (UFC) coverage defined in [77] and described in Section 3.1. We use the NuSMV bounded model checker along with the formal model of the system to automatically generate test cases providing requirements UFC coverage. The test generation technique is identical to the one described in 4.1 except we use the system model rather than the requirements model for test case generation.

A test suite thus generated will be highly redundant, as a single test case will often satisfy several UFC obligations. We therefore reduce this test suite using a greedy approach. We randomly select a test case from the test suite, check how many UFC obligations are satisfied and add it to a reduced test set. Next, we randomly pick another test case from the suite and check whether any additional UFC obligations were satisfied. If so, we add the test case to the reduced test set. This process continues till we have exhausted all the test cases in the test suite. We

now have a randomly reduced test suite that maintains UFC coverage over the LTL requirements. We generate three such reduced UFC test suites for each case example in our experiment to eliminate the possibility of skewing our results with an outlier (an extremely good or bad reduced test suite).

MC/DC over model:

The full test suite to provide MC/DC used in this experiment is the same one used in previous work [62]. We used the test suite that provides MC/DC over the inlined model (one in which the intermediate variables and function calls are inlined) rather than the non-inlined model as it is more rigorous and effective. We thus compare the requirements UFC coverage against a rigorous notion of MC/DC. The test suite was automatically generated using the NuSMV [55] model checker to provide MC/DC over the model. The full test suite was naïvely generated, with a separate test case for every construct we need to cover in the model. This straightforward method of generation results in highly redundant test suites, as with UFC test suite generation. Thus, the size of the complete test suite can typically be reduced while preserving coverage.

The approach to reduce the test suite is similar to that used for UFC coverage. As before, we generate three such reduced test suites to decrease the chances of skewing our results with an outlier (very good or very bad reduced test suite).

Requirements UFC Coverage + MC/DC over model:

To generate test suites providing both requirements UFC coverage and MC/DC over the model, we simply merge the test suite providing UFC with the test suite providing MC/DC. As mentioned previously, we generated three reduced MC/DC suites and three reduced UFC suites. It is thus possible to create nine different combined suites,

by merging each of the three reduced MC/DC suites with each of the three reduced UFC suites. Nevertheless, we elected to use only the best reduced MC/DC suite (with respect to fault finding among the reduced MC/DC suites) for creating the combined test suites since we were interested in determining if the UFC suites improved the fault finding of the best rather than average MC/DC suite. We thus merge the best MC/DC suite with each of the three reduced UFC suites to create only three combined suites.

5.1.2 Mutant Generation

To create mutants or faulty implementations, we built a fault seeding tool that can randomly inject faults into the implementation. Each mutant is created by introducing a single fault into a correct implementation by mutating an operator or variable. The fault seeding tool is capable of seeding faults from different classes. We seeded the following classes of faults:

Arithmetic: Changes an arithmetic operator $(+, -, /, *, mod, exp)$.

Relational: Changes a relational operator $(=, != , <, >, <=, >=)$.

Boolean: Changes a boolean operator (OR, AND, XOR).

Negation: Introduces the boolean NOT operator.

Delay: Introduces the delay operator on a variable reference (that is, use the stored value of the variable from the previous computational cycle rather than the newly computed value).

Constant: Changes a constant expression by adding or subtracting 1 from int and real constants, or by negating boolean constants.

Variable Replacement: Substitutes a variable occurring in an equation with another variable of the same type.

To seed a fault from a certain class, the tool first randomly picks one expression among all possible expressions of that kind in the implementation. It then randomly determines how to change the operator. For instance to seed an arithmetic mutation, we first randomly pick one expression from all possible arithmetic expressions to mutate, say we pick the expression 'a + b'; we then randomly determine if the arithmetic operator '+' should be replaced with '-' or '*' or '/' and create the arithmetic mutant accordingly. Our fault seeding tool ensures that no duplicate faults are seeded.

In our experiment, we generated mutants so that the 'fault ratio' for each fault class is uniform. The term fault ratio refers to the number of mutants generated for a specific fault class versus the total number of mutants possible for that fault class. For example, assume an implementation consists of R Relational operators and B Boolean operators. Thus there are R possible Relational faults and B possible Boolean faults. For uniform fault ratio, we would seed x relational faults and y boolean faults in the implementation so that $x/R = y/B$.

We generated three sets of 200 mutants for each case example. We generated multiple mutant sets for each example to reduce potential bias in our results from a mutant set that may have very hard (or easy) faults to detect. Our mutant generator does not guarantee that a mutant will be semantically different from the original implementation. Nevertheless, this weakness in mutant generation does not affect our results, since we are investigating the relative fault finding of test suites rather than the absolute fault finding.

The fault finding effectiveness of a test suite is measured as the number of mutants detected (or 'killed') to the total number of mutants created. We say that a mutant is detected by a test suite when the test suite results in different observed values between the mutant and the oracle implementation. The system model serves as the oracle implementation in conformance testing. We only observe the output values of the

model and mutants for comparison. We do not use internal state information. The reason being internal state information between the model and implementation may differ and can therefore not be compared directly. Additionally, in general, internal state information of the system under test may not be available and it is therefore preferable to perform the comparison with only output values.

5.2 Experimental Results

For each of the case examples—DWM_1, DWM_2, Vertmax_Batch, Latctl_Batch—we generated three reduced UFC test suites, three reduced MC/DC test suites, three combined UFC + MC/DC test suites and three sets of mutants. As mentioned earlier the combined suites are created by merging the best reduced MC/DC suite with each of the three reduced UFC suites. For this reason we compare the fault finding ability of the combined suites only against the best MC/DC suite rather than all the reduced MC/DC suites. We ran every test suite against every set of mutants, and recorded the percentage of mutants caught. For each case example, this yielded nine observations each for MC/DC, UFC and the combined test suites. We average the percentage of mutants caught across the mutant sets for each case example and each kind of test suite. This yields three averages, one each for MC/DC, UFC, and combined test suites as summarized in Table 5.1. Also, for each case example we identify the most effective MC/DC suite (among the generated three reduced MC/DC suites) and calculate average fault finding across the mutant sets. The 'Best MC/DC' column in the table represents these averaged observations. Additionally Table 5.1 also gives relative improvement in average fault finding of UFC suites over MC/DC test suites, and combined suites over the best MC/DC suite. Note that some of the numbers in the relative improvement column in Table 5.1 are negative. This implies that the test suite did not yield an improvement, instead did worse than the MC/DC test suite at

fault finding. For instance, for the DWM_1 model the MC/DC test suites provide an average fault finding of 84.6% and the UFC suites provide an average fault finding of 82.7%, and thus the relative improvement in fault finding for UFC suites is negative (= -2.2%) with respect to MC/DC suites. On the other hand, for the DWM_1 system, the combined suites provide better fault finding (an average of 91.5%) than the best MC/DC suite (85.8%), giving a positive relative improvement of 6.6%.

	Avg. MC/DC	**Avg. UFC**	**Rel. Improv.**	**Best MC/DC**	**Avg. Combined**	**Rel. Improv.**
DWM_1	84.6%	82.7%	-2.2%	85.8%	91.5%	6.6 %
DWM_2	90.6%	16.7%	-81.6%	90.6%	90.6%	0.0%
Latctl	85.1%	88.7%	4.2%	85.4%	94.6%	10.8%
Vertmax	86.0%	68.6%	-20.2%	86.0%	89.7%	4.3%

Table 5.1: Average percentage of mutants caught by test suites and relative improvement over MC/DC.

The complete set of 27 fault finding observations for each case example is presented in Table 5.2. Note that in the table, $MU1$, $MU2$, $MU3$ denote the three mutant sets; M_1, M_2, M_3 refer to the reduced MC/DC suites; U_1, U_2, U_3 to reduced UFC suites; and finally C_1, C_2, C_3 to the combined UFC and MC/DC suites. The best MC/DC suite (used to create combined suites) can be identified by comparing the fault finding of M_1, M_2, and M_3 across the mutant sets. Thus from the results in Table 5.2 we find for the DWM_1 system, the best MC/DC test suite is M_3. For DWM_2 and Vertmax_Batch systems, all three reduced MC/DC suites are equally effective so any of them can be used for creating the combined suites. We randomly selected M_1 for DWM_2 and M_3 for Vertmax_Batch system. Finally, for the Latctl_Batch system, M_1 is the most effective.

	DWM_1								
	M_1	**M_2**	**M_3**	**U_1**	**U_2**	**U_3**	**C_1**	**C_2**	**C_3**
MU1	82.7%	81.2%	84.3%	79.2%	79.7%	76.1%	88.3%	89.3%	88.8%
MU2	83.8%	83.8%	86.3%	83.8%	82.7%	81.2%	91.4%	91.9%	91.4%
MU3	86.3%	86.3%	86.8%	87.3%	87.8%	86.8%	93.9%	94.4%	94.4%
Size	73	76	77	463	469	468	540	546	545
	DWM_2								
	M_1	**M_2**	**M_3**	**U_1**	**U_2**	**U_3**	**C_1**	**C_2**	**C_3**
MU1	91.4%	91.4%	91.4%	17.2%	15.2%	16.7%	91.4%	91.4%	91.4%
MU2	91.4%	91.4%	91.4%	16.7%	14.6%	16.7%	91.4%	91.4%	91.4%
MU3	88.9%	88.9%	88.9%	18.7%	16.2%	18.7%	88.9%	88.9%	88.9%
Size	452	452	448	33	32	31	485	484	483
	Latctl_Batch								
	M_1	**M_2**	**M_3**	**U_1**	**U_2**	**U_3**	**C_1**	**C_2**	**C_3**
MU1	85.2%	84.2%	84.7%	89.3%	87.8%	89.8%	94.4%	92.9%	91.8%
MU2	85.7%	85.7%	85.2%	89.3%	89.3%	89.8%	96.4%	95.9%	95.9%
MU3	85.2%	84.7%	85.2%	88.8%	85.2%	88.8%	94.9%	93.9%	94.9%
Size	73	71	73	50	49	53	123	122	126
	Vertmax_Batch								
	M_1	**M_2**	**M_3**	**U_1**	**U_2**	**U_3**	**C_1**	**C_2**	**C_3**
MU1	83.8%	83.8%	83.8%	67.5%	69.5%	66.0%	88.8%	88.8%	88.8%
MU2	81.3%	81.3%	81.3%	71.1%	71.1%	71.6%	91.9%	90.4%	90.4%
MU3	88.9%	88.9%	88.9%	64.5%	66.0%	70.1%	87.3%	86.3%	88.3%
Size	301	299	297	89	79	88	386	376	385

Table 5.2: Complete results for all case examples.

From the results in Tables 5.1 and 5.2, it is evident that for all case examples, except the Latctl_Batch system, MC/DC test suites outperform the UFC suites in fault finding. The degree to which MC/DC suites are better, however, varies by a vast range. The maximum difference is on DWM_2, where MC/DC suites provide an average fault finding of 90.6% in contrast to 16.7% provided by UFC suites. The minimum difference is on DWM_1 where MC/DC provides an average fault finding of 84.6% versus 82.7% provided by UFC suites. The combined suites on the other hand outperform the MC/DC suites. The relative improvement provided by the combined suites however spans a much smaller range (0 - 10.8%). In other words, the number of different faults revealed by the UFC suites as compared to the best MC/DC suite is in the range of $0 - 10.8\%$ of the mutants seeded. The combined suites provide better fault finding than the best MC/DC suite on three of the four case examples. On the DWM_2 system the combined suites yield no improvement. A detailed discussion of the implication of these results is presented in Section 5.3.

5.2.1 Statistical Analyses

In this section, we statistically analyze the results in Tables 5.1 and 5.2 to determine if the hypotheses, H_1 and H_2, stated previously in the introduction of this chapter are supported.

To evaluate H_1 and H_2, we formulate our respective null hypotheses $H0_1$ and $H0_2$ as follows:

$H0_1$: A test suite generated to provide requirements UFC coverage will find the same number of faults as a test suite generated to provide MC/DC coverage over the model.

$H0_2$: A test suite generated to provide both requirements UFC coverage and

MC/DC over the model will reveal the same number of faults as a test suite generated to provide only MC/DC over the model.

To accept H_2, we must reject $H0_2$. Rejecting $H0_2$ implies that the data for the combined test suite and MC/DC suite come from different populations. In other words, this implies that either the combined suites have more fault finding than the MC/DC suites or vice versa. However, the combined suite includes the MC/DC suite and can therefore never have lesser fault finding than the MC/DC suite. This implies that H_2 is supported when $H0_2$ is rejected. On the other hand, rejecting $H0_1$ does not necessarily imply H_1 is supported, as this implies that the UFC suites have *different* fault finding ability than the MC/DC suites, not necessarily better fault finding ability. To accept H_1 after rejecting $H0_1$, we examine the data in the table and determine if the UFC suites have greater fault finding than the MC/DC suite. If so, we accept H_1. If the data indicates that UFC suites instead have lesser fault finding than the MC/DC suites, we reject H_1.

Our experimental observations are drawn from an unknown distribution, and we therefore cannot reasonably fit our data to a theoretical probability distribution. To evaluate $H0_1$ and $H0_2$ without any assumptions on the distribution of our data, we use the permutation test, a non-parametric test with no distribution assumptions. When performing a permutation test, a reference distribution is obtained by calculating all possible permutations of the observations [25, 46]. To perform the permutation test, we restate our null hypotheses as:

$H0_1$**:** The data points for percentage of mutants caught using the UFC and MC/DC test suites come from the same population.

$H0_2$**:** The data points for percentage of mutants caught using the MC/DC and combined UFC + MC/DC test suites come from the same population.

We evaluate the two hypotheses for each of the case examples. The procedure for permutation test of each hypothesis is as follows. Data is partitioned into two groups: A and B. Null hypothesis states that data in groups A and B come from the same population. We calculate the test statistic S as the absolute value of the difference in the means of group A and B:

$$S = \mid \overline{A} - \overline{B} \mid$$

We calculate $Number\ of\ Permutations$ as the number of ways of grouping all the observations in A and B into two sets. We then let $COUNT$ equal the number of permutations of A and B in which the test statistic is greater than S. Finally, $P - Value$ is calculated as:

$$P - Value = COUNT\ /\ Number\ of\ Permutations$$

For each case example, if $P - Value$ is less than the α value of 0.05 then we reject the null hypothesis with significance level α.

The null hypotheses $H0_1$ and $H0_2$ are evaluated using different groups of data. For $H0_1$, data for each case example in Table 5.2 is partitioned into two groups with nine observations each: % of faults caught by UFC test suites (group A – columns U_1, U_2, U_3 in the table), and % of faults caught by MC/DC test suites (group B – columns M_1, M_2, M_3). We calculate the $Number\ of\ Permutations$ as:

$$Number\ of\ Permutations = \binom{18}{9} = 48620$$

For $H0_2$, data for each case example in Table 5.2 is partitioned into two groups, one with nine observations and the other with three observations: % of faults caught by combined UFC+MC/DC test suites (group A – columns C_1, C_2, C_3), and % of faults caught by the best MC/DC suite (group B – M_1 column for DWM_2 and

Latctl_Batch systems, and M_3 column for DWM_1 and Vertmax_Batch systems). We calculate the $Number\ of\ Permutations$ as:

$$Number\ of\ Permutations = \binom{12}{9} = 220$$

We then determine the p-value for each hypothesis using the procedure described previously. Table 5.3 lists the p-values for both null hypotheses ($H0_1$ and $H0_2$) and states if the corresponding original hypotheses (H_1 and H_2) are supported for each case example. As mentioned earlier, for each case example, H_1 is supported if $H0_1$ is rejected with significance level $\alpha = 0.05$ and all the UFC suites (columns U_1, U_2, U_3 in Table 5.2) have better fault finding than the MCDC suites (columns M_1, M_2, M_3), and H_2 is supported if we reject $H0_2$.

	P-Value		**Result**	
	$H0_1$	$H0_2$	H_1	H_2
DWM_1	0.24	0.004	Unsupported	Supported
DWM_2	0.00004	1.0	Unsupported	Unsupported
Latctl_Batch	0.0002	0.004	Supported	Supported
Vertmax_Batch	0.00004	0.027	Unsupported	Supported

Table 5.3: Hypotheses Evaluation for different case examples

Given the p-values in Table 5.3 and the fault finding data in Table 5.2 we examine why the original hypotheses (H_1 and H_2) are supported/rejected for each case example. For the DWM_1 system, $H0_1$ is accepted (since p-value is greater than α value), and we therefore reject H_1. For the other three systems, $H0_1$ is rejected but the UFC suites outperform the MC/DC suites only on the Latctl_Batch system. For the DWM_2 and Vertmax_Batch systems, MC/DC suites always outperforms the UFC test suites. Thus, H_1 is supported on the Latctl_Batch system and rejected

on the DWM_2 and Vertmax_Batch systems. On the other hand, $H0_2$ is rejected (p-value less than the α value) on all but the DWM_2 system. This implies that H_2 is supported on all except the DWM_2 system. Thus, we find that with statistical significance level $\alpha = 0.05$ hypothesis H_1 is supported only on one case example, and hypothesis H_2 is supported on three of the four case examples.

5.2.2 Threats to Validity

While our results are statistically significant, they are derived from a small set of examples, which poses a threat to the generalization of the results.

Our fault seeding method seeds one fault per mutant. In practice, implementations are likely to have more than one fault. However, previous studies have shown that mutation testing in which one fault is seeded per mutant draws valid conclusions of fault finding ability [4].

Additionally, all fault seeding methods have an inherent weakness. It is difficult to determine the exact fault classes and ensure that seeded faults are representative of faults that occur in practical situations. In our experiment, we assume a uniform ratio of faults across fault classes. This may not reflect the fault distribution in practice. Finally, our fault seeding method does not ensure that seeded faults result in mutants that are semantically different from the oracle implementation. Ideally, we would eliminate mutants that are semantically equivalent, however, identifying such mutants is infeasible in practice.

5.3 Discussion

In this section we analyze and discuss the implications of the results in Tables 5.1 and 5.2. We present the discussion in the context of Hypotheses 1 and 2 stated in the introduction of this Chapter.

5.3.1 Analysis - Hypothesis 1

	Avg. MC/DC Achieved by UFC suites	**Achievable MC/DC**	**Rel. Diff.**
DWM_1	78.2%	92.5%	15.5%
DWM_2	25.8%	100%	74.2%
Latctl_Batch	88.6%	98.0%	9.6%
Vertmax_Batch	80.9%	99.8%	18.9%

Table 5.4: MC/DC achieved by the reduced UFC suites over the system model

As seen from Table 5.1, on all but one of the industrial systems, test suites generated for requirements UFC coverage have lower fault finding than test suites providing MC/DC over the system model. Statistical analysis revealed that hypothesis 1 stating "test suites providing requirements UFC coverage have better fault finding than test suites providing MC/DC over the model" was supported only on the Latctl_Batch system and rejected on all the other systems at the 5% significance level. We believe this may be because of one or both of the following reasons, (1) The UFC metric used for requirements coverage is not sufficiently rigorous and we thus have an inadequate set of requirements-based tests, and (2) Requirements are not sufficiently defined with respect to the system model. Thus, test suites providing requirements coverage will be ineffective at revealing faults in the model since there are behaviors in the model not specified in the requirements.

To assess the rigor of the UFC metric and the quality of the requirements with regard to behaviors covered in the system model, we measured MC/DC achieved by the reduced UFC suites over the system model. The results are summarized in Table 5.4. We found that for all the case examples, UFC test suites provide less than Achievable MC/DC over the system model. Thus, faults seeded in these uncovered

portions of the model cannot be revealed by the UFC suites. The extent to which the model is covered is an indicator of the effectiveness of the UFC metric and the quality of the requirements set. On the DWM_1, Vertmax_Batch, and Latctl_Batch systems the UFC suites do reasonably well, achieving an average MC/DC of 78.2%, 88.6%, and 80.9% respectively as compared to 92.5%, 98% and 99.8% achievable MC/DC. Note, however, that relative differences in MC/DC need not correspond exactly to relative differences in fault finding between the UFC and MC/DC suites (as seen in our examples). In addition to coverage, fault finding is also highly influenced by the nature and number of faults seeded in covered and uncovered portions of the model. The relation between coverage and fault finding is not the focus of this report and we hope to investigate this in our future work.

On the DWM_2 system, the UFC suites do poorly in both fault finding and MC/DC achieved. The UFC suites only achieve an average of 25.8% MC/DC over the model when compared to an achievable MC/DC of 100%. Correspondingly, the UFC suites have very poor fault finding (average of 16.7%) when compared to the MC/DC suites (average of 90.6%), since faults seeded in the uncovered portions of the model cannot be revealed by the UFC suites. The terrible fault finding and MC/DC achieved by the UFC suites on the DWM_2 system was surprising since we knew the system had a good set of requirements. To gain better understanding we took a closer look at the requirements set and the UFC obligations generated from them. We found that many of the requirements were structured similar to the sample requirement (formalized as an LTL property in the SMV [55] language) below,

```
LTLSPEC G(var_a > (
    case
        foo : 0 ;
        bar : 1 ;
```

```
    esac +
    case
        baz : 2 ;
        bpr : 3 ;
    esac
));
```

Informally, the sample requirement states that *var_a* is always greater than the sum of the outcomes of the two case expressions. When we perform UFC for the above requirement, it would result in obligations for the following expressions:

1. Relational expression within the globally operator (G)
2. Atomic condition foo within the first case expression
3. Atomic condition *bar* within the first case expression
4. Atomic condition *baz* within the second case expression
5. Atomic condition *bpr* within the second case expression

The above requirement may be restructured (to express the same behavior) so that the sum of two case expressions is expressed as a single case expression as shown:

```
LTLSPEC G(var_a > (
    case
        foo & baz : 0 + 2 ;
        foo & bpr : 0 + 3 ;
        bar & baz : 1 + 2 ;
        bar & bpr : 1 + 3 ;
    esac
));
```

Achieving UFC coverage over this restructured requirement will involve more obligations than before since the boolean conditions in the case expression are more complex. UFC would result in obligations for the following expressions in this restructured requirement:

1. Relational expression within the globally operator (G)
2. Complex condition foo & baz within the case expression
3. Complex condition foo & bpr within the case expression
4. Complex condition bar & baz within the case expression
5. Complex condition bar & bpr within the case expression

Thus, the structure of the requirements has a significant impact on the number and rigor of UFC obligations and hence the size of the test suite providing UFC coverage. In our experiment, we did not restructure requirements similar to the sample requirement discussed and instead retained the original structure. Therefore, the UFC obligations generated were fewer and far less rigorous. We believe this is the primary reason for the poor performance (both fault finding and MC/DC achieved) of the UFC suites for the DWM_2 system. The experience with the DWM_2 system suggests that even with a good set of requirements, rigorous requirements coverage metrics, such as the UFC metric, can be easily cheated since they are highly sensitive to the structure of the requirements. The issue here is similar to the sensitivity of the MC/DC metric to structure of the implementation observed in [62]. MC/DC was found to be significantly less effective when used over an implementation structure with intermediate variables and non inlined function calls as opposed to an implementation with inline expanded intermediate variables and function calls. Thus, as with all structural coverage metrics, we must be aware that the *structure* of the object used in measurement plays an important role in the effectiveness of the metrics.

To summarize, we find that the fault finding effectiveness of test suites providing requirements UFC coverage is heavily dependent on the nature and completeness of the requirements. Additionally, the rigor and robustness (with respect to requirements structure) of the requirements coverage metric used plays an important role in the effectiveness of the generated test suites. Thus, even with a good set of requirements, test suites providing requirements structural coverage may be ineffective if the coverage metric can be cheated. In our experiment, the UFC metric gets cheated when requirements are structured to hide the complexity of conditions on the DWM_2 system. Based on these observations and our results, we do not recommend using requirements coverage in place of model coverage as a measure of adequacy for conformance test suites.

5.3.2 Analysis - Hypothesis 2

As seen in Tables 5.1 and 5.2, for three of the four industrial case examples the combined UFC and MC/DC suites outperform the MC/DC suite in fault finding. For the DWM_2 system, however, the combined suites yield no improvement in fault finding over the MC/DC suite. Statistical analysis on the data in Table 5.2 revealed that Hypothesis 2 is supported with a significance level of 5% for the DWM_1, Vertmax_Batch, Latctl_Batch systems, and rejected for the DWM_2 system since the combined suites yield no improvement.

For the the DWM_1, Vertmax_Batch, Latctl_Batch systems, the combined UFC and MC/DC suites yielded an average fault finding improvement in the range of (4.3% - 10.8%) over the best MC/DC suite. The relative improvement implies that the UFC suites find a considerable number of faults not revealed by the best MC/DC suite.

To confirm that the improvement seen in DWM_1, Vertmax_Batch, Latctl_Batch

	Avg. UFC Achieved by MC/DC suites	**Achievable UFC**	**Rel. Diff.**
DWM_1	28.3%	96.9%	70.8%
DWM_2	59.7%	64.0%	6.7%
Latctl_Batch	94.7%	99.5%	4.8%
Vertmax_Batch	97.4%	99.0%	1.6%

Table 5.5: UFC achieved by the reduced MC/DC suites over the system model

systems is a result of combining the MC/DC metric with the UFC metric and not solely because of the increased number of test cases in the combined suites, we decided to measure the UFC coverage achieved over the requirements by the MC/DC suite. The results are summarized in Table 5.5. To understand the implications of the results in the table, consider the following two situations. If the MC/DC suite provides 100% achievable UFC over the requirements, it implies that the combined MC/DC + UFC coverage is satisfied by simply using the MC/DC suite instead of the combined suites. Under such circumstances, the fault finding improvement observed on combining the test suites would be solely due to the increased number of test cases. On the other hand, if the MC/DC suite provides less than achievable UFC over the requirements, it implies that there are scenarios/behaviors specified by the requirements that are not covered by the MC/DC suite but covered by the UFC suite. Thus, the combination may have proved more effective because of these additional covered scenarios and not simply because of increased test cases. We now take a closer look at the results in Table 5.5 to see which of these situations occurred. We found that in all three systems (Latctl_Batch, Vertmax_Batch, and DWM_1), the MC/DC suites provided less than achievable UFC coverage over the requirements. This indicates that the UFC suites

cover several behaviors specified in the requirements that are not covered by the MC/DC suite. We postulate that these additional covered behaviors contribute to the improved fault finding observed with the combined suites on these systems.

For the DWM_2 system, the combined suites yield no improvement in fault finding over the MC/DC suite, implying that the faults revealed by the UFC suites are a subset of the faults revealed by the MC/DC suite. The DWM_2 system consists almost entirely of complex Boolean mode logic, and the MC/DC metric is extremely effective for such kind of systems. There is thus a distinct possibility that the MC/DC suite reveals all the seeded faults (excluding semantically equivalent faults that can never be revealed). This belief was strengthened when we ran the full rather than reduced MC/DC and UFC suites and measured fault finding. We found that even with the full test suites, that have dramatically larger number of test cases, the combination did not yield any improvement in the number of faults revealed. Therefore, we believe that there is a strong possibility the MC/DC suites revealed all except the semantically equivalent faults on the DWM_2 system. Under such a circumstance, no test suite complementing the MC/DC suite can improve the fault finding thereby always rejecting Hypothesis 2. Such occurrences are anomalous and we discount them from our analysis.

To summarize, we found that for three of the four case examples, the combined test suite providing both requirements UFC coverage and MC/DC over the model is significantly more effective than a test suite solely providing MC/DC over the model. For the DWM_2 system that did not support this, we strongly believe that the MC/DC suites revealed all possible faults making improvement in fault finding on combining with UFC suites impossible. We disregard this abnormal occurrence to conclude that combined test suites have better fault finding than the MC/DC suites for all the systems. Given our results, we believe using requirements coverage metrics,

such as UFC, in combination with model coverage metrics, such as MC/DC, yields a significantly stronger adequacy measure than simply covering the model.

Note that for all the case examples, all three kinds of test suites—MC/DC, UFC, and combined—never yield 100% fault finding. This is because some of the seeded faults may result in mutant implementations that are semantically equivalent to the correct implementation (i.e., faults that *cannot* result in any observable failure). Thus, in each of our case examples, although we seed 200 faults giving 200 mutated implementations, it may be that less than 200 of the mutations are semantically different from the correct implementation and can therefore be detected. This is a common problem in fault seeding experiments [4, 56]. In industrial size examples it is extraordinarily expensive and time consuming, or—in most cases—infeasible to identify mutations that are semantically equivalent to the correct implementation and exclude them from consideration. Therefore, the fault finding percentage that we give in our experiment results is a conservative estimate, and we expect the actual fault finding for the test suites to be higher if we were to exclude the semantically equivalent mutations. Nevertheless, this issue will not affect our conclusions since we only judge based on relative fault finding rather than absolute fault finding.

5.4 Summary

Presently in model-based development, adequacy of conformance test suites is inferred by measuring structural coverage achieved over the model. In this chapter we investigated the use of requirements coverage as an adequacy measure for conformance testing. Our empirical study revealed that on three of the four industrial case examples, our hypothesis stating "Requirements coverage (UFC) is more effective than model coverage (MC/DC) when used as an adequacy measure for conformance test suites" was rejected at 5% statistical significance level. Nevertheless, we found

that requirements coverage is useful when used in combination with model coverage to measure adequacy of conformance test suites. Our hypothesis stating that "test suites providing both requirements UFC coverage and MC/DC over the model is more effective than test suites providing only MC/DC over the model" was supported at 5% significance level on three of the four case examples. The relative improvement in fault finding yielded by the combined suites over the MC/DC suites was in the range of $4.3\% - 10.8\%$. The system that did not support the hypothesis was an outlier where we firmly believe the MC/DC suite found all possible faults, making improvement with the combined suites impossible. Based on our results, we believe that the effectiveness of adequacy measures based solely on model coverage can certainly be improved. Combining existing metrics for model coverage and requirements coverage investigated in this report may be one possible way of accomplishing this. There may be other approaches, for instance, defining a new metric that accounts for both requirements and model coverage. Presently, however, in the absence of such a metric, *we highly recommend combining existing metrics for rigorous model coverage and requirements coverage to measure adequacy of conformance test suites.*

Another observation gained in our experiment relates to the sensitivity of requirements coverage metrics such as UFC to the structure of the requirements. Test suites providing requirements coverage may be ineffective even with an excellent set of requirements. This can occur when structure of the formalized requirements effectively "cheat" the requirements coverage metric. The UFC metric in our experiment was cheated when requirements were structured to hide the complexity of conditions in them. In our future work, we hope to define requirements coverage metrics that are more robust to the structure of the requirements.

Chapter 6

Related Work

We present related work as two sections; Section 6.1 describes work related to defining requirements coverage metrics, their use as a test adequacy metric, and as a means for measuring requirements completeness; Section 6.2 describes work related to requirements-based testing, i.e. , deriving test cases from requirements.

6.1 Related Work – Requirements Coverage

The work in this dissertation is related to work assessing the completeness and correctness of formulae in temporal logics. The most similar work involves *vacuity checking* of temporal logic formulas [7, 45, 60]. Intuitively, a model M *vacuously satisfies* property f if a subformula ϕ of f is not necessary to prove whether or not f is true. Formally, a formula is vacuous if we can replace ϕ by any arbitrary formula ψ in f without affecting the validity of f:

$$M \models f \quad \equiv \quad M \models f[\phi \leftarrow \psi]$$

Beer et al. [7] show it is possible to detect whether a formula contains vacuity by checking whether each of its atomic subformulas can be replaced by 'true' (if the subformula is of positive polarity)or 'false' (if the subformula is of negative poalrity) without affecting the validity of the original formula (the choice also depends on the structure of the formula and whether it is satisfied or not). To place it in our terms, this check determines whether each atomic condition independently affects the

formula in question. Using this notion of vacuity, they classify the formulas as either non-valid, vacuously valid, or non-vacuously valid. Traditionally, a formula is shown to be valid with a proof, and a formula that is non-valid is demonstrated by means of a counter-example using a standard model checker. Beer et al. show that a valid formula is non-vacuous by means of an *interesting witness*. An interesting witness is similar to a counter-example but proves non-vacuity, while a counter-example proves non-validity. They have formalized the notions of vacuity, and interesting witness, and show how to detect vacuity and generate interesting witness in temporal model checking.

Their approach in generating witness counterexamples for each atomic condition to prove non-vacuity is similar to the trap properties for UFC that are described in Section 3.2. Nevertheless, the goal of this work is quite different than ours. The purpose of performing vacuity detection on a formula over a model is to see whether or not a valid formula can be replaced by a stronger valid formula. This stronger formula may indicate problems within either the formula or the model. Our work is concerned with adequately testing requirements, potentially in the absence of a model. The complete vacuity check defined in [7] is one possible metric for assessing the adequacy of a test set. It is simpler and more rigorous metric than our UFC metric when used for test generation. In future work, we plan to reformulate the metric in [7] to support 'partial' weakening (as described in Section 3.1.4) and investigate its effectiveness.

Vacuity detection defined in [45] is an extension of the vacuity detection and interesting witness generation performed by Beer et al. [7]. Beer et al. identify a subset of ACTL called w-ACTL for which they perform vacuity detection, whereas in this paper, they generalize it to specifications of CTL^*. They also give a more rigorous definition of vacuity, checking whether all subformulas of the specification

affect the truth value in the system. An efficient implementation of the algorithm proposed in [45] is presented by Purandare and Somenzi [60] for all CTL formulae.

In our case study we examined how well coverage of requirements mapped to coverage of an implementation model. This is similar to recent research assessing the completeness of LTL properties over models. Chockler et al. propose coverage metrics for formal verification based on coverage metrics used in hardware simulation [19]. Mutations, capturing the different metrics of coverage, are applied to a given design and the resultant mutant designs are examined with respect to a given specification. Each mutation was generated to check whether a particular element of the design was essential for the satisfaction of the specification. Mutations correspond to omissions and replacements of small elements of the design. Two coverage checks were performed on the mutant design: *falsity coverage* (does the mutant still satisfy the specification?), and *vacuity coverage* (if the mutant design still satisfies the specification, does it satisfy it vacuously?). The following coverage metrics were defined for model checking based on the metrics from simulation-based verification:

Syntactic coverage: Coverage is measured over the syntactic representation of the design. Code coverage, circuit coverage, and hit count were the syntactic coverage metrics defined.

Semantic coverage: Strengthening of existing semantic mutation coverage metrics were proposed. In addition to satisfaction of the specification, they checked vacuous satisfaction as well on a mutant design. Finite State Machine (FSM) coverage (state coverage and path coverage), and assertion coverage were the semantic coverage measures defined.

Symbolic algorithms to compute the different types of coverage were proposed. In [17, 18], Chockler et al. propose additional metrics to determine whether all parts

of a model are covered by requirements. They consider specifications given as LTL formulas, or by automata on infinite words. They define the notions of node, structure, and tree coverage on an infinite tree t_F obtained by unwinding the FSM F of the design. Node coverage of a state w corresponds to flipping the value of an output variable q in one occurrence of w in the infinite tree. Structure coverage corresponds to flipping the value of q in all the occurrences of w in the tree. Tree coverage generalizes node and structure coverage and corresponds to flipping the value of q in some occurrences of w in the tree. They describe two symbolic algorithms to compute node, structure, and tree coverage (with minor changes to capture the different types of coverage). The first algorithm is built on top of automata-based model-checking algorithms. The second algorithm reduces the coverage problem to the model-checking problem. Our goal is to create a coverage metric that when provided a robust set of formal requirements and a test suite satisfying the metric will yield a high level of coverage of an implementation using standard coverage metrics. Chockler's work provides a more direct (and potentially accurate) assessment of the adequacy of the requirements. The process of building the set of mutant specifications and re-checking the properties is very expensive, however, and may not be feasible in practice.

Hoskote at. al [42] also defined a coverage metric to estimate the completeness of a set of properties against a design. The metric identifies the portion of the state space of the design that is covered by the properties. In each property, the authors identify a signal or proposition as the *observed signal* in that property. The coverage metric measures the coverage of properties with respect to this observed signal. According to their definition, a *covered set* of states for an observed signal is the set of reachable states in which the values of the observed signal must be checked to prove satisfaction of the property. The authors present a recursive algorithm to compute coverage for properties specified over a subset of ACTL. The coverage estimator is also capable of

generating traces to specific uncovered states.

As Jayakumar et al. stated [43], vacuity detection discussed in [7] and coverage estimation are complementary techniques. Vacuity detection is concerned with improving individual properties, while the latter is concerned with augmenting a set of properties so that, collectively, they represent a better specification. In the case studies conducted in [43], the authors found that detecting vacuity in properties before estimating their coverage resulted in better coverage and less work.

Unlike the requirements coverage metrics we proposed, the metric presented in [43] is based on the design model. Thus, it is capable of identifying functionality in the model not addressed in the properties, but it cannot point out missing functionality in the model. Thus, it is possible to achieve 100% coverage and still have an incomplete model. Secondly, the coverage metric is based on states, and not paths. The requirements coverage metric that we presented measures coverage over execution paths rather than states. Ultimately, any testing approach is interested in actual executions of the model over time and therefore coverage over execution paths. A state may be reached via several execution paths, and any of those paths will cover that state. Nevertheless, the behavior along that execution path is not verified.

Lakehal et al. [48] defined structural coverage criteria for LUSTRE/SCADE specifications. Lustre is a data-flow synchronous language, that describes the behavior of systems through the use of nodes, modules that are computed in a dataflow order to describe the behavior of a system. The paper proposes structural coverage criteria for specifications in LUSTRE/SCADE. Lustre specifications can be represented as operator networks, a labeled graph connecting operators by means of directed edges. The authors propose three coverage criteria of varying rigor based on paths in the operator networks. *Basic Coverage* (BC) requires every path of the program to be exercised once. *Elementary Conditions Coverage* (ECC) requires paths with a boolean input

to be exercised with both logical values of the input. *Multiple Condition Coverage* (MCC) requires both boolean inputs and internal edges to be set to all logical values. The MCC criterion is close to the constraints imposed by MC/DC.

The authors defined the structural coverage criteria with the aim of measuring coverage over designs modeled in Lustre/SCADE. Nevertheless, as observed in the paper, Lustre can express temporal properties and thus it is possible to specify the required properties of the system also in Lustre. Thus, coverage criteria defined in this paper can also be potentially used as requirements coverage criteria. In our work, we defined requirements coverage metrics over requirements formalized as Linear Temporal Logic properties. Nevertheless, as we had mentioned earlier, the notion of requirements coverage is not restricted to requirements formalized as LTL properties, and can be applied to any other formal notation. Lustre is one such formal notation. The work in [48] is thus very relevant to requirements coverage, and the metrics defined can be used directly to measure coverage over requirements formalized as Lustre specifications.

In the context of conformance testing, we conducted empirical investigations that indicated combining structural coverage metrics over the implementation with requirements coverage metrics served as a more effective test adequacy measure than simply using structural coverage metrics over the implementation. Briand et al. found similar results in their empirical study [14], though in the context of state-based testing for complex component models in object-oriented software. Combining a state-based testing technique for classes or class clusters modeled with statecharts [31], with a black-box testing technique, category partition testing, proved significantly more effective in fault detection.

Our investigation differs from the one performed by Briand et al. in several respects. We generate test suites providing requirements UFC coverage for black-box

testing. UFC test suites exercise execution paths in the model as opposed to states exercised by category partition testing. Additionally, all the test suites used in our study are automatically generated to mirror recent practices in model-based development where test suite generation tools are gaining rapid popularity. The investigation conducted by Briand et al. uses test cases manually generated by graduate students in a laboratory setting. Also, our domain of concern is slightly different. All the examples we considered are industry sized systems from the critical systems domain. Briand et al. use C++ and Java classes from a Domain Name Server and an academic software system in their experiments.

Reasoning with Temporal Logic on Truncated Paths

Eisner et al. [23] define semantics for reasoning with LTL on finite truncated paths. This work is very useful in our definition of coverage measurement of finite test case over LTL properties. The paper defines three different semantics for temporal operators: weak, neutral, and strong. The traditional LTL semantics defined over finite paths by Manna et al. [49] is the neutral view. 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. The strong and weak semantics are a coupled dual pair because the negation operator switches between them. The semantics defined in this paper are provided as variant re-formulations of the neutral semantics in [49]. In our definition of UFC coverage, we provide two views: a neutral view that uses the semantics of LTL over finite paths defined by Manna et al. [49], and the weakened view which uses a combination of the neutral and weak semantics presented by Eisner et al.

Runtime Verification Vs Requirements Coverage

We believe the concept of runtime monitoring/verification will be useful in measuring requirements coverage. In runtime monitoring/verfication a software component monitors the execution of a program and check its conformity with a requirement specification. Therefore, we can use obligations for the desired requirements coverage criteria to monitor the execution traces through the system and coverage can be computed based on the number of monitor formulas that were satisfied. Goldberg et al. [28] have developed a language, called EAGLE, that allows behavioral properties to be expressed over time. Properties in LTL can be embedded as rules in EAGLE. The authors have developed a framework that allows temporal properties expressed in EAGLE to be monitored over a system. Thus, requirements coverage obligations expressed in LTL can be expressed as monitors in EAGLE. The main disadvantage in using the runtime monitoring approach to measure requirements coverage lies in the need for a system to generate execution traces on which to monitor execution. Thus the benefit of using requirements coverage in measuring adequacy of test suites early in the development process without have to wait for an implementation is lost if we use the runtime monitoring approach. Nevertheless, in situations where an implementation is available, runtime monitoring offers a viable alternative for measuring requirements coverage.

6.2 Related Work – Requirements-Based Testing

Tan et al. [72] use the vacuity check presented in [7] to define a property coverage metric over LTL formulas. Their coverage metric could potentially be used as a measure of adequacy of black box test suites. However, Tan et al. use the metric as a means of test case generation from requirements properties. They consider system requirements formalized as LTL properties and assume that the model of the system

in the form of Kripke structures is available. The property coverage metric is identical to the vacuity check presented in [7]. The property coverage criteria is defined as:
ST is a property-coverage test suite for a system T_s and an LTL property f if T_s passes ST and ST covers every subformula
A test t covers a subformula ϕ of f if there is a mutation $f[\phi \leftarrow \psi]$ such that every Kripke structure T that passes t will not satisfy the formula $f[\phi \leftarrow \psi]$. This is the definition of non-vacuity given by Beer et al. Using this idea, the test cases are generated from the properties as follows:

- Each LTL property is transformed into a set of ∃LTL formulae called trapping formula. For every subformula ψ (or for every atomic proposition as defined by Beer et al.) of a property f, they generate a ∃LTL formula of the form:
  $\neg\ f[\psi\ \leftarrow\ (\tau)]$
  where $\tau = false$ if ψ has positive polarity in f
  and $\tau = true$ if ψ has negative polarity in f
  which is a mutation of f in which ψ is replaced by true or false depending on its polarity.

- These ∃LTL formula can be treated as trap properties and fed to the model checker to generate witnesses as tests that satisfy the property coverage criteria.

- The infinite tests produced by the model checker is truncated to a finite test case.

Their test truncating strategy relies on the fact that the tests generated by a model checker are lasso-shaped. In a black-box test setting, they assume knowledge of the upper-bound n on the number of states. To truncate the lasso-shaped tests to a finite one in a black-box test setting, they create a finite prefix of the lasso-shaped test by repeating the loop part of the test n times. In a white-box test setting, the finite-prefix

is constructed by tracking the states traversed and terminating whenever the same state has been visited twice at the same position on the loop. This methodology of generating acceptable black-box tests is highly impractical since most of the systems used in practice have very large n, therefore repeating the loop part of the test n times might not be feasible. The goal of our work in defining structural coverage metrics based on requirements that allow us to measure the adequacy of black-box test suites is different from theirs. They propose an approach to generate black-box test suites that satisfy the property coverage metric and enable the testing of linear temporal properties on the implementation. Also, Tan et al. [72] do not have any notion of test weakening and their notion of acceptable black box tests, while rigorous, is not practical for the reason stated previously about their test truncating strategy. In addition, the efficacy of the proposed metric is not explored.

In [1], Abdellatif-Kaddour et al. propose an approach that uses the specification of a property to drive the testing process. The objective of the approach is to find a system state where the target property if violated. The proposed approach constructs test scenarios in a step wise fashion. Each step explores possible continuations of "dangerous" scenarios identified at the previous step. A dangerous situation is described as one in which the system reaches intermediate states from which property violations may eventually occur if the system is not robust enough to recover. The proposed approach is based on the assumption that the system will gradually evolve towards critical failure, traversing states of dangerous situations. A random sampling technique is used to select test sequences at each step.

This approach differs from our proposed approach in that the test case construction to detect property violations uses the specification of the system. This testing approach is not implementation independent and will thus not be beneficial in early bug detection. In addition, the test case construction is guided by the property

and is *not directly derived* from the property. Therefore, the constructed test cases will not be directly traceable to the properties. The approach also requires manual intervention especially in identifying the dangerous situations.

In [26], Fraser et al. present a notion called *property relevance*, that helps determine if there is a relation between a test case and a property. This notion helps provide traceability of a test case to a property; thus, if the implementation were to fail a test case, the property relevance notion helps determine if it violates a property for which the test case is relevant. The authors present a method for measuring property relevance achieved by test cases. Additionally, the authors define a criteria that combines property relevance with structural coverage criteria over the model. They also present an approach using model checkers to automate test case generation for this combined criteria.

The goal of this work is similar to ours, to establish traceability between test cases and requirement properties, and to measure coverage achieved by test cases over requirements. Nevertheless, the property relevance criteria is too weak since it only checks if the property is relevant to the test case. It does not determine whether the property is exercised by test cases in several interesting ways. As mentioned earlier and explored in [7, 45, 60], one can come up with a naive test case satisfying a property vacuously. Such a test case will not be useful in fault finding. It is therefore important to ensure that the test case exercises the property in interesting ways. Additionally, to establish property relevance of a test case, the authors propose to create a model of the test case and then model check the property on this test case model. With potentially thousands of test cases and hundreds of requirements properties, this approach would necessitate model checking to be performed more than a hundred thousand times. This approach is not practical given the time needed for each model checking effort. Also, the test case generation approach for property relevance has

high complexity. To create positive test cases that are property relevant using model checkers, the approach requires the correct model to be combined with a mutated model and test case generation performed on this combined model. Combining models will significantly increase the difficulty of using model checkers especially on industrial systems with large state spaces.

Winbladh et al. [47] explored manually deriving requirements-based tests using requirements in the forms of goals and scenarios. The goal of this approach is to validate how well the system meets the requirements, and to identify weaknesses in the requirements early in the development process. Test scenarios are derived by following particular paths in the scenarios. The event outputs from the system are compared against the events in the requirements scenarios to validate the system using the test case. When this comparison is not possible, the authors propose to use the run time pre and post conditions in the requirements scenario to validate the system. Goals, plans, and scenarios are used to describe stake holder goals at different levels of abstraction. GoalML—an XML-based language—is used to express goals and plans. Scenarios—refinements of goals—are expressed using ScenarioML.

The objective of the approach in [47] is similar to our work in requirements-based testing, namely to validate the system using test cases derived directly for requirements and to identify weaknesses in the requirements. As opposed to this work, we use a more formal approach based on requirements formalized as temporal logic properties. Additionally, we automate the generation of requirements-based tests.

The last piece of related work that we present is the Reactis tool from Reactive Systems Inc. It is a commercially available test case generation tool that can be used to perform requirements-based testing. 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. 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 did some preliminary experiments with the Reactis tool for requirements-based test generation on some small models to examine the scalability of the tool [76]. We found that the test generation facility in Reactis does not scale linearly with the size of the model. The model coverage achieved over

models with a significant number of non-linear constraints over floating-point variables was also poor. However, these are preliminary results over a few models; more experimentation is needed to generalize these results.

Chapter 7

Conclusions

In this dissertation, we introduced the notion of requirements coverage and defined potential metrics that could be used to assess requirements coverage. There are several potential benefits of defining requirements coverage metrics, including:

1. a *direct measure* of how well a black-box test suite (or validation test suite) addresses a set of requirements,

2. an *implementation independent* assessment of the adequacy of a suite of black-box tests,

3. a means for measuring the *adequacy of requirements* on a given implementation,

4. a formal framework that, given a model, allows for *autogeneration* of tests that are immediately traceable to requirements

5. a measure of *adequacy of conformance tests* that test conformance of an implementation to a design

We defined three coverage metrics in Chapter 3, namely: requirements, coverage, requirements antecedent coverage and requirements UFC coverage, at different levels of rigor, over the structure of requirements formalized as Linear Temporal Logic (LTL) properties. We conducted an empirical study comparing the effectiveness of the three different metrics on a close to production model of a flight guidance system. Chapter 3 also discusses how to adapt the definitions for the metrics so they can

measure finite test cases.

We observed in our empirical study that given a well defined set of requirements and a rigorous testing metric such as the requirements UFC coverage, we achieve a high level of coverage of an implementation of the requirements. We found that a requirements-based test suite can be used to help determine the completeness of a set of requirements with respect to an implementation, and a test suite which yields high requirements coverage and low implementation coverage illustrates one of three problems:

1. The implementation is incorrect. The implementation allows behaviors not specified by the requirements. Hence a test suite that provides a high level of requirements coverage will not cover these incorrect behaviors, thus resulting in poor model coverage.

2. There are missing requirements. Here, the implementation under investigation may be correct and more restrictive than the behavior defined in the original requirement; the original requirements are simply incomplete and allow behaviors that should not be there. Hence we need additional requirements to restrict these behaviors. These additional requirements necessitate the creation of more test cases to achieve requirements coverage and will, presumably, lead to better coverage of the implementation.

3. The criteria chosen for requirements coverage is too weak.

In Chapter 4, we proposed, implemented and evaluated an approach that automates the generation of requirements-based tests for software validation. Our approach uses requirements formalized as LTL properties. We developed a framework that uses model checkers for auto generation of test cases. Test cases were generated

through an abstract model, that we call the requirements model, to provide requirements coverage over the properties. We evaluated our approach using three realistic examples from Rockwell Collins Inc.: prototype model of the mode logic from a flight guidance system, and two models related to the display window manager system. We measured coverage achieved by the generated tests on the system model. We found our approach was feasible with regard to time taken and number of test cases generated for all three systems. The generated tests achieved high coverage over the system model for the display window manager systems that had a well defined set of requirements. On the other hand, the tests generated from the requirements of the flight guidance system covered the system model poorly since the flight guidance system had an incomplete set of requirements. Based on our empirical investigation we believe our proposed approach to generating requirements-based tests provides three benefits:

1. Saves time and effort when generating test cases from requirements.

2. Effective method for generating validation tests when the requirements are well defined.

3. Helps in identifying missing requirements and over-constrained designs/models.

In Chapter 5, we investigated using requirements coverage metrics as an adequacy measure for conformance testing in the model-based software development approach. Currently, adequacy of conformance test suites is assessed by measuring coverage achieved over the model. Adequacy measured in this manner does not give a direct measure of how well the implementation exercises the requirements. In our investigation, we combine existing adequacy metrics measuring coverage over the model with a requirements coverage metric that we define in this dissertation. We conducted empirical studies that investigated the effectiveness of the combined metrics. More

specifically, we investigated the hypothesis stating that "test suites providing both requirements UFC coverage and MC/DC over the model is more effective than test suites providing only MC/DC over the model". Effectiveness refers to the fault finding capability of the generated test suites. We developed tools that allowed a large number of mutants to be randomly and automatically seeded in the systems and ran the test suites against the correct and mutated systems to assess fault finding effectiveness. We found that our hypothesis was supported at 5% statistical significance level on all realistic case examples. Tests providing requirements coverage found several faults that remained undetected by tests providing model coverage (in this case MC/DC coverage) making the combination more effective. Based on the empirical data in our experiments, we recommend that future measures of conformance testing adequacy consider both requirements and model coverage either by combining existing rigorous metrics, such as MC/DC and requirements UFC, or by defining new metrics that account for both behaviors in the requirements as well as the model.

In sum, we strongly encourage the software engineering community to consider the notion of requirements coverage as a test adequacy metric in software development. The requirements coverage metrics defined in this dissertation only provide a start for the rigorous exploration of metrics for requirements-based testing. We have merely defined the notion and explored the application and feasibility of the metrics on a few industrial examples. There are several topics that require further study:

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. Finding a formalism and notation that is acceptable to practicing developers, requirements engineers, and domain experts is necessary. In our work we have used LTL, but we are convinced that there are notations better suited to the task at hand.

Requirements coverage criteria: To our knowledge, there has been little other work on defining coverage criteria for high-level software requirements. Therefore, we do not know what coverage criteria will be useful in practice. In this dissertation, we conducted empirical studies that investigated the feasibility and effectiveness of requirements coverage metrics using a small number of industrial case examples. Presently, there is lack of industrial systems with completely formalized and a well defined set of requirements. In the future, however, given the trend towards formal methods in domains such as critical avionics, we hope that many more such systems will be available. We hope to conduct a more extensive empirical investigation and determine a requirements coverage criteria that (1) help us assess test suites with regard to their effectiveness in fault finding in implementations and (2) do not require test suites of unreasonable size.

Requirements versus model coverage: We must explore the relationship between requirements-based structural coverage and model or code-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? Investigating the relationship between requirements coverage and model coverage will help assess whether requirements have been sufficiently defined for the system.

Test case generation method: We believe the test case generation tool or technique used can have a significant effect on the effectiveness of the generated requirements-based test suites. We plan to evaluate the effect that different test case generation methods have on the fault finding capability and model

coverage achieved by test suites that provide a high level of structural coverage of the requirements. We plan to investigate a variety of model checkers and their strategies in this regard.

In sum, we believe that the notion of coverage metrics for requirements-based testing holds promise and we look forward to exploring each of these topics in future investigations.

Appendix A

Automated Test Generation

It is possible for functional testing to be fully automated when specifications are given in terms of some formal model. Several research efforts have developed techniques for automatic generation of tests from the formal models. Our research group has previously explored using model checkers to generate tests to provide structural coverage of formal models [37, 66, 67, 65]. The formal modeling notations used in our case examples and the automated test generation tools that this dissertation uses are discussed in the following sections.

A.1 Formal Modeling Notations

SCADE : The SCADE Suite from Esterel Technologies is a mature, commercially-supported toolset that is tailored to building reactive, safety-critical systems. In SCADE, software is specified using a combination of hierarchical state machines and block diagrams. The SCADE graphical notation is built on top of the kernel language Lustre [30]. In our research, we support automated test case generation for systems modeled in SCADE/Lustre. Lustre is a synchronous dataflow language that describes the behavior of systems through the use of nodes, modules that are computed in a dataflow order to describe the behavior of a system. This syntax can be graphically represented as block diagrams that are often used to describe requirements for control engineering. The dataflow model has several merits:

- It is a completely functional model without side effects. This feature makes the model well-suited to formal verification and program transformation. It also facilitates reuse, as a module will behave the same way in any context into which it is embedded.

- It is a naturally parallel model, in which the only constraints on parallelism are enforced by the data-dependencies between variables. This allows for parallel implementations to be realized, either in software, or directly in hardware.

Because of the syntax of Lustre and the checks of the compiler, the composition of these nodes always yields a complete, deterministic function. Thus, there is only one possible interpretation of a SCADE model. The SCADE notation is natural for describing dataflow systems, but can be difficult to use for describing control logic. To address this deficiency, SCADE is now bundled with Esterel Studio. This allows Esterel models to be embedded as nodes into SCADE models to describe this functionality. SCADE is bundled with a graphical editor for Lustre specifications, a graphical simulator, a model checker, a document generation tool, and several code generators, one of which is qualified to DO178B level A. The code generated by SCADE is suitable for use in safety-critical environments.

Simulink and Stateflow : Simulink [51] is a platform for multidomain simulation and Model-Based Design for dynamic systems. It provides an interactive graphical environment and a customizable set of block libraries, and can be extended for specialized applications. Simulation capabilities for both discrete and continuous systems are provided. Simulink has an enormous number of companion products and add-ons. These include test case generators, coverage analyzers, and code generators. The Simulink notation is natural for describing data-driven, control-law

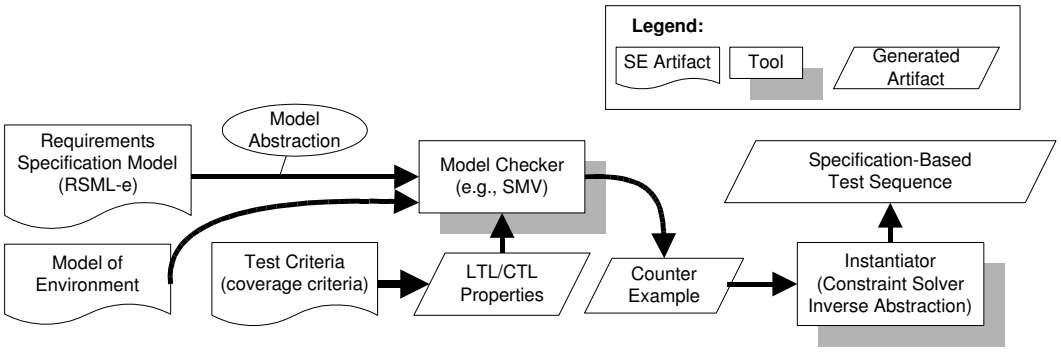

Figure A.1: Test sequence generation overview and architecture.

functionality of systems, but can be difficult for describing mode logic. Therefore, Simulink has a companion product, Stateflow [50], which can be used to describe mode logic. Stateflow is an interactive design and simulation tool for event-driven systems. Stateflow provides the language elements required to describe complex logic in a natural, readable, and understandable form. It is tightly integrated with MATLAB and Simulink, providing an efficient environment for designing embedded systems that contain control, supervisory, and mode logic. Mathworks supports a plug-in code generation tool for Simulink models called Real-Time Workshop. This tool provides a wide range of code generation options, and has good traceability back to the original specification, and good code style. There are also third party code generators that have been used to autogenerate safety-critical avionics systems. Simulink is widely used in the aerospace industry to design, simulate, and autocode software for avionics equipment. We support automated test case generation for systems modeled in Simulink and Stateflow.

A.2 Model Checkers as Automated Test Generation Tools

Model checking is an algorithmic technique for checking temporal properties of systems modeled as finite state machines. Model checkers exhaustively explore the reachable state space of the model searching for violations of the properties under investigation [22]. Should a property violation be detected, the model checker will produce a counter-example illustrating how this violation can take place. For instance, if the property under consideration is a safety property (that basically specifies what should happen or not happen), a counter-example would be a sequence of inputs that will take the finite state model from its initial state to a state where the safety property is violated.

A model checker can be used to find test cases by formulating a test criterion as a verification condition for the model checker. For example, we may want to test a transition (guarded with condition C) between states A and B in the formal model. We can formulate a condition describing a test case testing this transition—the sequence of inputs must take the model to state A; in state A, C must be true, and the next state must be B. This is a property expressible in the logics used in common model checkers, for example, the logic LTL. We can now challenge the model checker to find a way of getting to such a state by negating the property (saying that we assert that there is no such input sequence) and start verification. We call such a property a *trap property* [27]. 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 exercise the transition of interest. By repeating this process for each transition in the formal model, we use the model checker to automatically derive test sequences that will give us transition coverage of the model. Obviously, this general approach can be used to generate tests for a wide variety of structural coverage criteria, such as all state variables have take on every

value, and all decisions in the model have evaluated to both true and false. The test generation process is outlined in Figure A.1.

The approach discussed above is not unique to our group, several research groups are actively pursuing model checking techniques as a means for test case generation [2, 3, 27, 41, 66, 40]. We use the the NuSMV [55] model-checker in our research to automatically generate test cases that provide requirements coverage. The NuSMV model checker has been designed to be an open architecture for model checking, which can be reliably used for the verification of industrial designs, as a core for custom verification tools, as a testbed for formal verification techniques, and applied to other research areas . Properties to be verified in NuSMV are specified using either branching time logic (CTL) or linear time logic (LTL). A discussion on how we use model checkers to automatically generate test cases that provide requirements coverage is provided in Section 4.1.

Bibliography

[1] O. Abdellatif-Kaddour, P. Thvenod-Fosse, and H. Waeselynck. Property-Oriented Testing: A Strategy for Exploring Dangerous Scenarios. In *Proceedings of the 2003 ACM symposium on Applied computing*, 2003.

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

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

[4] J.H. Andrews, L.C. Briand, and Y. Labiche. Is Mutation an Appropriate Tool for Testing Experiments? *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 402–411, 2005.

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

[6] R. Armoni, D. Bustan, O. Kupferman, and M. Y. Vardi. Aborts vs. Resets in Linear Temporal Logic. In *TACAS*, pages 65–80, November 2003.

[7] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient Detection of Vacuity in ACTL Formulas. In *Formal Methods in System Design*, pages 141–162, 2001.

[8] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 2nd edition, 1990.

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

[10] B. Bezier. *Software Testing Techniques, 2nd Edition*. Van Nostrand Reinhold, New York, 1990.

[11] R. Bharadwaj and C. Heitmeyer. Model Checking Complete Requirements Specifications Using Abstraction. In *First ACM SIGPLAN Workshop on Automatic Analysis of Software*, 1997.

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

[13] B. Boehm. *Software Engineering; R&D Trends and Defense Needs.* MIT Press, Cambridge, MA, 1979.

[14] L.C Briand, M. Di Penta, and Y. Labiche. Assessing and Improving State-Based Class Testing: A Series of Experiments. *IEEE Transactions on Software Engineering*, 30 (11), 2004.

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

[16] 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.

[17] H. Chockler, O. Kupferman, R. P. Kurshan, and M. Y. Vardi. A Practical Approach to Coverage in Model Checking. In *Proceedings of the International Conference on Computer Aided Verification (CAV01), Lecture Notes in Computer Science 2102*, pages 66–78. Springer-Verlag, July 2001.

[18] H. Chockler, O. Kupferman, and M. Y. Vardi. Coverage Metrics for Temporal Logic Model Checking. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science 2031*, pages 528–542. Springer-Verlag, April 2001.

[19] H. Chockler, O. Kupferman, and M. Y. Vardi. Coverage Metrics for Formal Verification. In *12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods, volume 2860 of Lecture Notes in Computer Science*, pages 111–125. Springer-Verlag, October 2003.

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

[21] E. M. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. *ACM Transaction on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

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

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

[24] A. Engels, L. M. G. Feijs, and S. Mauw. Test Generation for Intelligent Networks Using Model Checking. In *Proceedings of TACAS'97, LNCS 1217*, pages 384–398. Springer, 1997.

[25] R.A. Fisher. *The Design of Experiment*. New York: Hafner, 1935.

[26] G. Fraser and F. Wotawa. Property Relevant Software Testing with Model-Checkers. In *Second Workshop on Advances in Model-based Software Testing (A-MOST'06)*, 2006.

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

[28] A. Goldberg and K. Havelund. Automated Runtime Verification with Eagle. In *MSVVEIS*, 2005.

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

[30] 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.

[31] D. Harel and R. Marelly. *Come Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

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

[33] M.P.E. Heimdahl and G. Devaraj. Test-suite Reduction for Model Based Tests: Effects on Test Quality and Implications for Testing. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE)*, Linz, Austria, September 2004.

[34] M.P.E. Heimdahl, G. Devaraj, and R.J. Weber. Specification Test Coverage Adequacy Criteria = Specification Test Generation Inadequacy Criteria? In *Proceedings of the Eighth IEEE International Symposium on High Assurance Systems Engineering (HASE)*, Tampa, Florida, March 2004.

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

[36] M.P.E. Heimdahl, S. Rayadurgam, and W. Visser. Specification Centered Testing. In *Second International Workshop on Analysis, Testing and Verification*, May 2001.

[37] M.P.E Heimdahl, S. Rayadurgam, W. Visser, G. Devaraj, and J. Gao. Auto-Generating Test Sequences using Model Checkers: A Case Study. In *3rd International Worshop on Formal Approaches to Testing of Software (FATES 2003)*, 2003.

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

[39] B. Hetzel. *The Complete Guide to Software Testing*. John Wiley and Sons, 1988.

[40] H.S. Hong, S.D. Cha, I. Lee, O. Sokolsky, and H. Ural. Data Flow Testing as Model Checking. In *Proceedings of the International Conference on Software Engineering*, Portland, Oregon, May 2003.

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

[42] Y. Hoskote, T. Kam, P. Ho, and X. Zhao. Coverage Estimation for Symbolic Model Checking. In *Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 300–305, 1999.

[43] N. Jayakumar, M. Purandare, and F. Somenzi. Dos Don'ts of CTL State Coverage Estimation. In *Proceedings of the Design Automation Conference*, June 2003.

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

[45] O. Kupferman and M. Y. Vardi. Vacuity Detection in Temporal Model Checking. *Journal on Software Tools for Technology Transfer*, 4(2), February 2003.

[46] P.H. Kvam and B. Vidakovic. Nonparametric Statistics with Applications to Science and Engineering. 2007.

[47] K.Winbladh, T.Alspaugh, and D.Richardson. Meeting the Requirements and Living Up to Expectations. In *ISR Technical Report UCI-ISR-07-1, Dept. of Informatics, Univ. of California, Irvine*, 2007.

[48] A. Lakehal and I. Parissis. Structural Test Coverage Criteria for Lustre Programs. In *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, 2005.

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

[50] MathWorks. The MathWorks Inc. Corporate Web Page. Via the world-wide-web: http://www.mathworks.com, 2004.

[51] Mathworks Inc. Simulink Product Web Site. Via the world-wide-web: http://www.mathworks.com/products/simulink.

[52] S.P. Miller, E.A. Anderson, L.G. Wagner, M.W. Whalen, and M.P.E. Heimdahl. Formal Verification of Flight Critical Software. In *Proceedings of the AIAA Guidance, Navigation and Control Conference and Exhibit*, August 2005.

[53] S.P. Miller, M.P.E. Heimdahl, and A.C. Tribble. Proving the Shalls. In *Proceedings of FM 2003: the 12th International FME Symposium*, September 2003.

[54] S.P. Miller, A.C. Tribble, T. Carlson, and E.J. Danielson. Flight Guidance System Requirements Specification. Technical Report CR-2003-212426, NASA, June 2003.

[55] The NuSMV Toolset, 2005. Available at http://nusmv.irst.itc.it/.

[56] A.J. Offutt and J. Pan. Automatically Detecting Equivalent Mutants and Infeasible Paths. *Software Testing, Verification & Reliability*, 7(3):165–192, 1997.

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

[58] W.E. Perry. *A Standard for Testing Application Software*. 1990.

[59] M. Pezze and M. Young. *Software Test and Analysis: Process, Principles, and Techniques*. John Wiley and Sons, October 2006.

[60] M. Purandare and F. Somenzi. Vacuum Cleaning CTL Formulae. In *Proceedings of the 14th Conference on Computer Aided Design*, pages 485–499. Springer-Verlag, 2002.

[61] A. Rajan, M.W. Whalen, and M.P.E. Heimdahl. Model Validation Using Automatically Generated Requirements-Based Tests. In *Proceedings of the IEEE High Assurance Systems Engineering Symposium (HASE 2007)*, Novemeber 2007.

[62] A. Rajan, M.W. Whalen, and M.P.E. Heimdahl. The Effect of Program and Model Structure on MC/DC Test Adequacy Coverage. In *Proceedings of 30th International Conference on Software Engineering (ICSE)*, To appear in May 2008. Available at http://crisys.cs.umn.edu/ICSE08.pdf.

[63] A. Rajan, M.W. Whalen, M. Staats, and M.P.E. Heimdahl. Requirements Coverage as an Adequacy Measure for Conformance Testing. In *Proceedings of International Conference on Formal Engineering Methods (ICFEM)*, October 2008.

[64] S. Rayadurgam. *Automatic Test-case Generation from Formal Models of Software*. PhD thesis, University of Minnesota, November 2003.

[65] S. Rayadurgam and Mats P.E. Heimdahl. Generating MC/DC Adequate Test Sequences Through Model Checking. In *Proceedings of the 28th Annual IEEE/NASA Software Engineering Workshop – SEW-03*, Greenbelt, Maryland, December 2003.

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

[67] S. Rayadurgam and M.P.E. Heimdahl. Test-Sequence Generation from Formal Requirement Models. In *Proceedings of the 6th IEEE International Symposium on the High Assurance Systems Engineering (HASE 2001)*, Boca Raton, Florida, October 2001.

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

[69] D. J. Richardson, S. L. Aha, and T. O'Malley. Specification-based Test Oracles for Reactive Systems. In *Proceedings of the 14th International Conference on Software Engineering*, pages 105–118. Springer, May 1992.

[70] RTCA. *DO-178B: Software Considerations In Airborne Systems and Equipment Certification*. RTCA, 1992.

[71] I. Somerville. *Software Engineering*. Addison-Wesley, eighth edition, 2007.

[72] L. Tan, O. Sokolsky, and I. Lee. Specification-Based Testing with Linear Temporal Logic. In *IEEE Int. Conf. on Information Reuse and Integration (IEEE IRI-2004)*, November 2004.

[73] E. Weyuker, T. Goradia, and A. Singh. Automatically Generating Test Data from a Boolean Specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, May 1994.

[74] M.W. Whalen. A Formal Semantics for $RSML^{-e}$. Master's thesis, University of Minnesota, May 2000.

[75] M.W. Whalen. Autocoding Tools Interim Report. In *NASA Contract NCC-01-001 Project Report*, February 2004.

[76] M.W. Whalen and S.P. Miller. Autocoding Tools Final Report. In *NASA Contract NCC-01-001 Project Report*, November 2005.

[77] M.W. Whalen, A. Rajan, and M.P.E. Heimdahl. Coverage Metrics for Requirements-Based Testing. In *Proceedings of International Symposium on Software Testing and Analysis*, July 2006.