On the Effectiveness of Specification-Based Structural Test-Coverage Criteria as Test-Data Generators for Safety-Critical Systems

A DISSERTATION
SUBMITTED TO THE FACULTY OF UNIVERSITY OF MINNESOTA
BY

Devaraj George

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

Dr. Mats P.E. Heimdahl, Advisor
November, 2012
Acknowledgments

I would like to thank my graduate school advisor Professor Mats P.E. Heimdahl for introducing me to the fascinating area of research in the field of software engineering. He has been a source of guidance and support throughout my days at the University of Minnesota. Also, I would like to express my heartfelt thanks to all members of the Safety Critical Systems Group that I have been associated with, especially to Matt Staats who set up the software test infrastructure for conducting the expanded studies on test effectiveness. His help in this regard is very much appreciated.

I would like to thank our sponsors NASA Ames, NSF, and Rockwell Collins for giving us an opportunity to perform research in the area of specification-based testing and for providing us with industrial-strength experimental subjects for conducting our studies.

Last, but not the least, I would like to dedicate this dissertation to my Father Siluvaimuthu George and my late mother Soundaram George for loving and constantly motivating me in this lifetime.
Abstract

Specification based testing aims to reduce the cost of testing and increase the reliability of systems that are of a safety critical nature. A major benefit of a formal specification is the ability to automatically construct test sequences that can be executed on the implementation software. A significant portion of the testing effort in such systems requires us to demonstrate test coverage as mandated by regulatory agencies. Specification based structural test-coverage criteria that mimic source code based criteria offer us a way to analyze the structure of the specification. This typically requires coverage of certain elements such as states, branches, and decisions. These formal specifications also provide us with a means to automatically generate test sequences to satisfy test coverage. Since one of the goals of software testing is to demonstrate the existence of faults, selection of test sequences that can reveal faults is of paramount importance. Nevertheless, the relationship between test-coverage criteria and fault detection is not well established in testing literature.

In this dissertation, we investigate the effectiveness of test-coverage criteria when used to drive test-data generation in the safety-critical systems domain. We provide two core contributions. First, due to the lack of sufficient evidence in testing research regarding the quality of test sets generated to satisfy test-coverage criteria, we empirically evaluate the fault-finding ability of test-sets generated to various test-coverage criteria proposed in the testing literature. Second, we study the effect of test-suite reduction techniques on the generated test-data sets to empirically evaluate the sensitivity of test-coverage criteria to test-suite reduction techniques.

Our findings have raised serious doubts about the use of test-coverage criteria as test-data generators in this domain. In the initial studies conducted, test sequences
generated to these coverage criteria perform significantly worse at fault detection when compared to random testing that uses the same effort measured in terms of time to generate and run tests for structural test-coverage criteria such as transition coverage. In the expanded study which was conducted following the initial studies, we evaluated the fault detection effectiveness of test suites reduced to satisfy both branch and MC/DC coverage criteria against a reduced test suite of equal size using a set of random test data. The results from the expanded study validate our earlier findings and have provided us with solid statistical evidence confirming that satisfaction of a highly complex coverage criterion alone is a poor indication of test suite quality. The findings from our studies indicate a need for methods to determine test adequacy that not only provide the desired coverage, but also lend themselves as targets for automated test generation techniques. These criteria must address the problem holistically to account for all factors influencing the quality of testing, including the program structure, the nature of the state space of the system under test, the test oracle used, and finally, the test generation mechanism itself.

In addition, we find that reduction techniques designed to minimize the size of a test suite while maintaining structural coverage may significantly reduce the fault-finding effectiveness of the test suite.
Contents

List of Tables vii

List of Figures ix

1 Introduction 1

1.1 The Software Testing Problem ................................. 1
1.2 Specification-Based Testing ................................. 3
1.3 Objectives ..................................................... 7
1.4 Research Overview ............................................. 11
  1.4.1 Our Approach ............................................. 11
  1.4.2 Research Tasks ............................................ 13
1.5 Contributions ................................................... 16
1.6 Organization ..................................................... 18

2 Background and Related Work 20

2.1 Software Test Coverage (or Adequacy) Criteria ................. 20
2.2 Structural Test Coverage Criteria ................................ 21
  2.2.1 Code-Based Structural Test Coverage Criteria .......... 21
  2.2.2 Specification based Test Coverage Criteria .............. 25
2.3 Empirical Studies on Test Effectiveness ......................... 32
2.4 Empirical Studies on association between Test Reduction and Test Effectiveness ................................ 38
2.5 Chapter Conclusion ............................................. 40
# Automated Test Case Generation from Formal Models of Software

## 3 Automated Test Case Generation from Formal Models of Software

### 3.1 Formal Specification of Synchronous Reactive Systems

### 3.2 Synchronous Dataflow Languages

### 3.3 Specification-Based Testing

### 3.4 Model Checking

#### 3.4.1 Temporal Logics for Model Checking

#### 3.4.2 Computation Tree Logic

#### 3.4.3 Linear Time Temporal Logic

#### 3.4.4 Explicit State Model Checking

#### 3.4.5 Symbolic Model Checking

#### 3.4.6 Bounded Model Checking

### 3.5 Model Checking for Automatic Test-Data Generation

### 3.6 Chapter Conclusion

# Initial Study on Test Effectiveness

## 4 Initial Study on Test Effectiveness

### 4.1 Experimental Setup

#### 4.1.1 Case Example: The Flight Guidance System

#### 4.1.2 Fault Injection

#### 4.1.3 Test Set Generation and Execution

### 4.2 Experimental Results

### 4.3 Key Observations

### 4.4 Criteria with Execution-Guarantee

### 4.5 Formalization of the Execution-Guarantee Policy

### 4.6 Enhanced Coverage Criteria with Execution Guarantee

### 4.7 Fault Finding Experiment with Execution Guarantee

#### 4.7.1 Hypotheses

#### 4.7.2 Experimental Results and Analysis

### 4.8 Threats to Validity
<table>
<thead>
<tr>
<th>Page</th>
<th>Section Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>84</td>
<td>4.9 Chapter Conclusions</td>
<td></td>
</tr>
<tr>
<td>87</td>
<td>5 Expanded Study on Test Effectiveness</td>
<td></td>
</tr>
<tr>
<td>88</td>
<td>5.1 Experimental Setup</td>
<td></td>
</tr>
<tr>
<td>89</td>
<td>5.1.1 Experimental Hypothesis</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>5.1.2 Experimental Overview</td>
<td></td>
</tr>
<tr>
<td>92</td>
<td>5.1.3 Mutant Generation</td>
<td></td>
</tr>
<tr>
<td>94</td>
<td>5.1.4 Test Data Generation</td>
<td></td>
</tr>
<tr>
<td>94</td>
<td>5.1.5 Test Suite Reduction</td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>5.1.6 Computing Fault Finding</td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>5.2 Experimental Results and Key Observations</td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>5.3 Discussion of Results and Implications for Model-Based Testing</td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>5.4 Chapter Conclusion</td>
<td></td>
</tr>
<tr>
<td>106</td>
<td>6 Impact of Test Reduction on Test Effectiveness</td>
<td></td>
</tr>
<tr>
<td>106</td>
<td>6.1 Hypotheses</td>
<td></td>
</tr>
<tr>
<td>107</td>
<td>6.2 Experimental Setup</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>6.2.1 Fault Injection and Detection</td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>6.2.2 Test Set Generation and Reduction</td>
<td></td>
</tr>
<tr>
<td>114</td>
<td>6.3 Key Observations</td>
<td></td>
</tr>
<tr>
<td>115</td>
<td>6.3.1 Test-Suite Reduction</td>
<td></td>
</tr>
<tr>
<td>116</td>
<td>6.3.2 Effect on Fault Detection Effectiveness</td>
<td></td>
</tr>
<tr>
<td>119</td>
<td>6.4 Threats to Validity</td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>6.5 Chapter Conclusions</td>
<td></td>
</tr>
<tr>
<td>122</td>
<td>7 Conclusions and Future Work</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td>Bibliography</td>
<td></td>
</tr>
</tbody>
</table>
List of Tables

4.1 Number of trap properties generated for each coverage criterion .... 67
4.2 Average length of the test cases in the test-suites ................. 67
4.3 Number of faults revealed by each test-suite .................... 68
4.4 Inference rules for determining usage of atom “a” in a boolean expression “e” .................................................. 76
4.5 Syntax of terms and values ............................................ 77
4.6 Evaluation Rules for $e \mapsto e'$ ..................................... 77
4.7 Test set generation times (using NuSMV) and Fault detection effectiveness for various criteria ................................. 81

5.1 Case Example Information ............................................. 91
5.2 Average fault-finding effectiveness results, branch coverage criterion. OO = Output-Only, MX = Maximum ....................... 97
5.3 Average fault-finding effectiveness results, MCDC criterion. OO = Output-Only, MX = Maximum ................................. 97
5.4 Coverage Achieved (of Maximum Coverage) by Randomly Generated Test Suites Reduced to Satisfy Coverage Criteria .......... 98

6.1 Full test set generation for various criteria along with their fault detection capability .................................................. 114
6.2 Reduced test set sizes for various test reduction runs. r.1–r.5 indicate the size of the five reduced test-suites. Avg. denotes the average reduction of the five. .................................................. 116
6.3 Fault finding capability of the reduced test-sets. r.1–r.5 indicate the number of faults found with each of the five reduced test-suites. Avg. denotes the average number of faults found with the five reduced test-suites.
## List of Figures

1.1 The Specification-Based Testing Process ........................................ 4  
1.2 Framework for Test sequence generation for the language RSML$^{-e}$ ..................... 10  
1.3 Fault types used in Fault Seeding .............................................. 12  
1.4 Experimental Design For Mutation Testing ...................................... 13  
2.1 Structural Test Coverage Criteria Subsumption Hierarchy ......................... 26  
3.1 The Specification-Based Testing Process ........................................ 45  
4.1 Framework for Test sequence generation for the language RSML$^{-e}$ ..................... 61  
4.2 Flight Guidance System .......................................................... 64  
4.3 An example fault seeded into the FGS model. .................................. 65  
4.4 Fault seeding. ............................................................................ 65  
4.5 Experimental Design ................................................................. 66  
4.6 Code using a variable (a) and a function (b) and (c) to structure a larger condition. ................................................................. 71  
4.7 Enforcing Execution-Guarantee Policy ............................................ 75  
5.1 Experimental Design for Expanded Study ......................................... 92  
5.2 Caption for LOF ........................................................................... 99  
5.3 Caption for LOF ........................................................................... 100  
6.1 Experimental Design for Test Reduction Study ..................................... 108  
6.2 Algorithm for test-suite reduction.................................................... 113
Chapter 1

Introduction

1.1 The Software Testing Problem

Software testing is the process of executing a software system to determine whether it matches its specification and executes in its intended environment. The fact that the system is being executed distinguishes testing from static analysis techniques such as code reviews, in which uncompiled source code is read and analyzed statically. To plan and execute tests, software testers must consider the software and the function it computes, the inputs and how they can be combined, and the environment in which the software will eventually operate. One of the major limitation of software testing is that it can only show the presence of failures, not their absence. This is a fundamental, theoretical limitation; generally speaking, the problem of finding all failures in a program is undecidable [93]. Therefore, exhaustive testing of software is infeasible because of the complexity of software which is generally intractable. However, various techniques are used to make testing more rigorous, such as Functional testing [102, 105], which requires selection of test scenarios without regard to source code structure. In Functional testing, test selection methods and test adequacy criteria are based on the attributes of the specification or operational environment and not on attributes of the code or data structures (often referred to as black-box testing or specification-based testing) [77, 60]. The other commonly used testing technique is structural testing which requires that inputs be based solely on the structure of the source code or its data structures (white-box testing, program-based testing) [56, 105].
Regardless of the breakthroughs in static V&V techniques, such as model checking and theorem proving, testing is still an invaluable technique for Verification and Validation (V&V) that cannot be replaced. This is especially true in software development for high assurance systems, such as software controlling safety critical applications in aeronautics, space, medical devices etc. In such projects, however, testing is often a costly and time consuming process. For instance, the validation and verification phase (V&V) consumes approximately 50%-70% of the software development resources. Currently, the majority of V&V time is devoted to the development of test cases that adequately test the required functionality of the software as well as cover the implementation. Thus, if the process of deriving test cases for V&V could be automated and provide requirements-based and code-based test suites that satisfy the most stringent standards for critical systems (such as, for example, DO-178B—the standard governing the development of flight-critical software for commercial aviation), dramatic time and cost savings would be realized. Since most of these systems usually have a high level description of their functional behavior which is captured as a set of functional specifications, automatic generation of test cases from requirements specifications has found considerable interest in the research community [20, 41, 50, 108, 107, 19]. Such automation could result in dramatic time and cost savings, especially for verifying safety-critical systems. The use of explicit behavioral models is motivated by the observation that traditionally, the process of deriving tests tends to be unstructured, barely motivated in the details, not reproducible, not documented, and bound to the ingenuity of test engineers. The idea is that the existence of an artifact that explicitly encodes the intended behavior can help mitigate the implications of these problems. In this dissertation, we will be focusing on the use of such behavioral models of software to help mitigate the testing problem in the context of specification-based testing which we will explain in the following
1.2 Specification-Based Testing

Functional testing [16] is a widely used mechanism for deriving test cases, but both industrial practice and research are still far from general and satisfactory methodologies. Key reasons for this are the intrinsic difficulty of the problem and the difficulty of working with informal specifications.

An active research area concerns the use of formal methods for deriving test cases. In the past, formal methods have been mainly studied as a means for formally proving software properties [17, 111, 112, 113]. Recently, significant attention has been focussed on their use for deriving test cases [127, 21, 92, 20]. In this approach, the functional specification is expressed in a mathematical (or formal) notation, making it suitable to be manipulated by automated means. Since this functional specification is machine-readable and formal to the extent that it has a well-defined behavioral interpretation, test cases can in principle be derived mechanically. This formal specification represents the possible configurations of the system under test. To find test cases, the specification is searched for executable paths. A possible execution path can serve as a test case. For finding appropriate test cases, i.e. paths that refer to a certain test requirement, the search of the paths has to be guided.

Specification-Based testing is a variant of testing that relies on explicit behavioral models that encode the intended behavior of a system and possibly the behavior of its environment. Generally speaking, the specification-based testing process can be conveniently divided into the following five main steps as shown in Figure 3.1:

1. **Model** the program or system under test (SUT) and/or its environment. The first step of specification-based testing is to create a functional specification,
focus on what we want to test and omit many of the details of the system under test. Most formal notations provide some automated verification tools that allow us to explore the behavior of the specification and check the expected behavior.

2. Generate abstract tests from the model. We choose test selection criteria, to say which we want to generate from the formal specification. For example, we might interact with the test generation tool to choose a particular specification-based coverage criterion, such as transition coverage, or we might focus on a particular portion of the specification. The main output of this step is a set of abstract test cases which lack some of the detail needed by the SUT and are
not directly executable on it. The abstract tests, however, can be executed on a simulation environment to see the expected behavior of the system.

3. **Concretize** the abstract tests to make them executable. The third step of specification-based testing is to transform the abstract tests created in step 2 into executable concrete tests. This may be done by a transformation tool, which uses mappings to translate each abstract test case into an executable test script.

4. **Execute** the tests on the program or system under test (SUT) and obtain test results. The fourth step is to execute the concrete tests on the system under test. We can do through online means, where the tests are executed as they are produced, or through offline means, where the test scripts are executed after we generate all the concrete tests.

5. **Analyze** the test results. The fifth step is to analyze the results of the test executions and take corrective action. For each test case that reports a failure, we need to determine the fault that caused that failure.

Current research work has focussed on techniques for automatically deriving test cases from particular formal notations. Most of these techniques rely on the use of structural test-coverage criteria defined over the formal specification as a basis for generating test cases; a test-coverage criterion basically specifies a particular software test requirement and thereby helps determine the test inputs to satisfy that requirement. The structural test coverage criteria are specifically designed to exercise the structure of the formal specification in terms of both control (conditions, decisions, etc.) and data flow (testing definition-use associations).

However, one of the research challenges that the use of these formal techniques open up is the question of the suitability of these structural test-coverage criteria
for test-case generation, in other words – as test generation targets. In other words, although much work has been done in terms of automated test-case generation from formal specifications using structural test-coverage criteria, there have been limited studies on the efficacy of these generated tests in terms of their ability to detect realistic or “naturally” occurring faults in software [97, 96, 6]. For instance, structural test-coverage criteria based on control flow and data flow analysis have been widely studied in the testing literature and used as a basis for test-data generation from formal specifications. But there is no definitive answer to the question as to how effective these generated tests are at finding software faults. Therefore, data indicating how effective these techniques are in general would be useful for testing practitioners.

The goal of this dissertation is to evaluate test-coverage criteria defined over formal specifications of software and compare them with respect to their effectiveness and robustness in terms of fault detection. Mutation Analysis [97, 79, 76] would be used to assess and to compare the effectiveness of the different structural test-coverage criteria with respect to fault detection. Although, several variants of mutation analysis have been proposed in literature, we use the definition where a mutant is a single-point, syntactically correct change introduced in an implementation P that is derived from a specification S, and a set of mutants of S consist of a set of implementations that differ from S in containing one mutation from a given list of fault types, representative of the most likely faults committed by developers, when deriving a Program P from the specification S. Specification-Based structural test-coverage criteria are based on the – usually implicit – assumption that the derived program resembles its formal specification. Furthermore, its assumed that typical and interesting faults in the program correlate with, for example, misimplementing transitions in the specification or implementing arithmetic comparisons, which appear in a formal specification, incorrectly in the program. These considerations form the basis for
techniques such as mutation-based test assessment. The term “robustness” is used to mean the variation in effectiveness of the tests when subjected to test conditions such as, for example, test-set reduction. The results of such an evaluation will be useful in identifying a set of structural test coverage criteria that are suitable (in terms of both cost of test set generation and effectiveness in terms of fault finding) in the domain of formal specification-based testing. This dissertation specifically focusses on formal specifications modeled using a class of languages called synchronous dataflow languages \cite{55, 104, 14, 15} and test-coverage criteria defined over such formal specifications. A synchronous dataflow language is based on the synchrony hypothesis: the behavior of the program is a sequence of reactions, where each reaction consists of reading the current inputs, updating the internal state, and computing the current outputs. Languages such as Lustre \cite{54} fall in this category of synchronous data flow languages. Later sections in this report will discuss these types of languages in more detail. Such languages can be used to model safety-critical reactive systems \cite{59}, reactive systems are systems that have to continually respond to stimuli from the external environment.

In the next few sections, the overall goal of this dissertation and the significant research tasks that have been undertaken to accomplish the goals and the major contributions of this work will be highlighted.

### 1.3 Objectives

The goal of this dissertation is to evaluate and compare structural test-coverage criteria defined over formal specifications of software systems in terms of their usefulness at fault detection on implementations derived from the specification. As mentioned earlier, we will restrict the scope of our studies to the domain of safety-critical reactive systems. The evaluation will help us determine if these test-coverage criteria are
any good when used as a mechanism for test-data generation, i.e., as test-generation targets. In the process of performing this evaluation, we seek to address the following open research questions:

1. What is the correlation between structural test-coverage criteria and fault finding ability of the test-sets generated using those criteria? What are the costs involved?

2. What sort of structural test-coverage criteria defined over formal specifications are most successful in uncovering realistic faults in implementations derived from those specifications?

3. How robust are these test-coverage criteria to the use of test-suite reduction techniques? Are stronger coverage criteria more robust to the effects of test-set reduction than weaker coverage measures? Here, the term “robustness” is defined as the variation in fault finding capability between the full and the reduced test sets generated for a particular test-coverage criterion, where the full and reduced test sets are coverage adequate with respect to the coverage criterion in question.

The evaluation of various specification-based test-coverage criteria can be made in two ways: First, through analytical comparisons that show the theoretical relationships between various test-coverage criteria [45, 46]. In this way, one could make claims about these criteria that are true in all situations. Second, through empirical comparisons which show relations that are based on specific studies [97, 9]. Although it is difficult to show that empirical studies hold in all situations, analytical comparisons between these criteria cannot always be performed, but empirical comparisons can. For instance, most previous analytical comparisons of testing criteria have been based on the \textit{subsumes} relation and its variants [46, 45]. In those studies, criterion
$C_1$ subsumes criterion $C_2$, if for every program $P$ and specification $S$, every test suite that satisfies $C_1$ for $(P, S)$ also satisfies $C_2$. But even if we theoretically prove that $C_1$ subsumes $C_2$, it is possible that a $C_2$ adequate test suite for some $(P, S)$ detects a fault while a $C_1$ adequate test-suite fails to do so. This is so because there is no necessary connection between $C_1$ and finding faults; test inputs may satisfy the desired coverage, but we may not be choosing a test input that reveals a fault. In other words, the subsumes relation expresses nothing about the ability to expose faults or assure quality. Therefore, determining the effectiveness of these criteria cannot be performed by analytical means.

The objective of this dissertation is to evaluate and compare the various specification-based test-coverage criteria as test-generation targets, by designing empirical studies that investigate the usefulness of these test-coverage criteria in terms of their ability to detect realistic faults in implementations derived from those specifications.

We will be conducting our empirical studies in the context of a specification centered development approach. This approach is centered around the use of a formal model of the behavior of a system (i.e., a formal specification) to drive various development activities such as prototyping, simulation, analysis, testing, and code generation. Specification-based testing [96, 77, 60] where the formal model of the behavior of the software is used as the source artifact to generate test cases, has recently gained significant interest in the software engineering community and has been proposed as an complementary technique to code based testing which can help in reducing the cost of test generation, increasing the effectiveness of the tests, and shortening the testing cycle. For our purpose, we will be using the formal specification of the software to drive the generation of complete test sequences.

The test-set generation strategy we will use is based on model checking technology. Model checking [53, 25, 29, 68, 32] is an algorithmic technique for verification
of systems modelled as finite state machines and provides an automatic way to generate tests for a variety of test-coverage criteria. In this approach, the test-coverage criteria are expressed as properties in temporal logic, and the counter example generating capability of model checkers is leveraged to obtain a test-case that satisfies the test requirement encoded by the temporal property. Previous empirical studies have shown that these techniques scale well to large, industrially-sized systems [64, 63].

Figure 1.2 shows an example of a framework for specification-based test-set generation centered around a formal specification language called RSML$^{-e}$ [121]. RSML$^{-e}$ is a synchronous data flow language that is used to model reactive systems. In this framework, test-coverage criteria, such as transition coverage, are encoded as falsifiable temporal logic properties that will force the test-case generator (in this case the NuSMV model checker) to produce counter examples (test cases) that exercise a particular transition.
1.4 Research Overview

As mentioned earlier, the goal of our work is to assess how well in terms of fault finding structural coverage criteria perform as test case generation targets. In particular, we are interested in how well such generated test perform during conformance testing (sometimes called verification testing). In conformance testing, we are attempting to determine whether or not an implementation derived from a formal specification has been derived correctly. To make such a determination one can generate test cases from the formal specification, run them on the derived implementation, and see if the result from executing the test conforms with the behavior predicted by the specification.

In the following sections, we will outline our overall research methodology and the tasks we undertook to address our objectives.

1.4.1 Our Approach

Our approach has been to use a test framework discussed earlier in section 1.3 for automating the generation of test cases from formal specifications modeled in the RSML$^{-e}$ formalism [121, 123]. (In the last experiment described in this dissertation, however, we use a modified version of the framework operating on Stateflow [88, 57] models as opposed to RSML$^{-e}$.) We used the bounded model checking capability of the model checker NuSMV to generate coverage adequate test cases [62, 50].

We devised several case studies to address our research objectives. In the first case study [62], we conducted an experiment to evaluate the fault finding capability of three commonly suggested specification-based structural coverage criteria, state, transition, and decision coverage.

In the second case study [61], the aim was to determine how well a test-suite generated to provide a certain structural or condition based coverage of a specification
model reveals faults in the implementation of that model as compared to a reduced test-suite providing the same coverage of the original model. In both these case studies we used a model of the mode-logic in a production sized flight-guidance system (FGS), written in the RSML−e language as our experimental subject. The FGS was ideally suited for test-case generation using model checkers since it is discrete—the mode logic consists entirely of enumerated and Boolean variables.

In a final study constructed to generalize our findings form the first study, we expanded the scope of the evaluation of the fault finding ability of tests generated to provide branch and MC/DC coverage criteria and evaluated these criteria on five industrial examples. We chose branch coverage since it is commonly used in software testing research and MC/DC [26], because it is mandated in regulatory standards such as DO-178B [110].

We used mutation analysis to determine the fault detection effectiveness of test-suites auto generated to specification-based test-coverage criteria. To provide targets for our testing effort, we created a collection of faulty implementations (or mutants). We implemented a random fault seeder to inject representative faults to create a suite of faulty programs (see Figures 1.3 and 1.4). For the test oracle, we chose one that
observed all output and internal state variables to determine whether or not a mutant was killed.

1.4.2 Research Tasks

We undertook the following research tasks to accomplish our goal and answer the open research questions outlined in Section 1.3:

Empirically Measuring the Effectiveness of Specification-Based Structural Test-Coverage Criteria: Using the experimental setup described above we performed the experiment by conducting the following steps for state, transition, and decision coverage criteria:
1. We used the original specification to generate a test-suite to a coverage criterion of interest, for example, transition coverage, and measured the effort involved.

2. We ran the test-suite on the 100 faulty implementations that were generated using a random fault seeder and recorded the number of faults revealed as well as the time required run the test-suite.

3. We used the same effort (the sum of the time used to generate as well as run the structural test) to generate and run a randomly generated test-suite. We generated this suite using a statistical testing tool also implemented as part of the Nimbus Toolset.

4. Given the results of the previous steps, we compared the relative fault finding capability of the randomly generated tests versus the structural tests.

Our experiments indicated that the common specification test-adequacy criteria we evaluated are woefully inadequate and are not likely to reveal faults in an implementation derived from the specification. Tests generated to these criteria perform poorly when compared to random tests that were generated using the same time and effort we spent in generating the structural test-suites. We identified two reasons for this inadequacy: First, our test-case generation technique based on model checking relentlessly exploited the structure of our experimental subject to achieve desired coverage without actually exercising the true decision logic in the model. Second, and more importantly, when the criteria are used as test-case generators rather then test-case metrics issues taken for granted when instrumenting models and code, for example, that the program flow actually reaches the construct of interest, need to be made explicit [62].

---

1 In the initial study, our formalization of the test-coverage criteria for use with a model-checker was inaccurate because it did not require execution of the constructs of interest such as decisions in the code, an oversight which we addressed in subsequent studies.
In our subsequent study, we addressed the two inadequacies identified in our first study by, 1. Formalizing the notion of evaluation in our test obligations, and, 2. Ensuring that our test generation mechanism exercises the model without misusing the program structure of our case example. We reran our experiments including a much broader variety of structural test-coverage criteria for our study. The results indicated that structural tests exercising condition-based coverage criteria such as transition and decision coverage performed much better than in the previous studies. In this experiment, stronger condition-based test-coverage criteria such as MC/DC found all seeded faults in the implementation. We could not however generalize the results because we used only a single case example – although a production size one. To address this threat to the external validity of our studies, we performed an expanded study using more case examples and controlled for various test factors such as choice of test oracle, test generation mechanism, and program structure.

In the expanded study, we used the same experimental design used in the earlier studies to run our experiments, but we expanded the scope of our investigation to use more case examples, generated more mutant implementations per case example, and controlled the choice of test oracle used and the program structure. The results yielded one key conclusion – Coverage criteria satisfaction alone is a poor indicator of test-suite effectiveness. Effectiveness of Structural tests were impacted with the choice of the test oracle used and the structure of the program under test. The observations from this study point to the inherent dangers in the increase in test automation in critical systems and the urgent need for more research into how coverage criteria, generation approach, and system structure jointly influence test effectiveness.

**Empirically Measuring the Robustness of Specification-Based Structural Test-Coverage Criteria**

In this experiment the aim was to determine how well a test-suite generated to
provide a certain structural or condition based coverage of a specification model reveals faults in the implementation of that model as compared to a reduced test-suite providing the same coverage of the original model. We generated and ran five reduced test-suites for each full test-suite, ensuring that the desired coverage criterion was maintained. We generated five reduced sets for each full test-suite to avoid biasing our results in the selection of tests for a reduced test-suite. We used a simple greedy based heuristic algorithm to generate significantly reduced test-suites. We then recorded and compared the relative fault finding capability of the full test-suites versus the reduced test-suites [61].

Our experimental results indicated that test reduction results in significant savings in terms of test-suite size. However, test-suite reduction adversely impacts the fault finding ability of test-suites that are derived from synchronous data-flow models. From our results we also observed that the more rigorous coverage criteria, such as MC/DC [27, 28], seem to be the least sensitive to the effect of test-suite reduction. The experiment indicates that test-suite reduction of test-suites providing structural coverage may not be an effective means of reducing testing effort—the cost is terms of lost fault finding capability is simply too high; especially in the safety-critical domain in which we are mainly interested.

1.5 Contributions

In this dissertation we make the following specific contributions towards understanding the effectiveness and ultimately, the usefulness of test-coverage measures when used for the purposes of test-data generation:

**Empirical Evidence on Test Effectiveness:** We have formally defined, evaluated, and compared a collection of specification-based structural test-coverage criteria
with respect to their effectiveness in terms of finding faults in implementations derived from such specifications. Through our initial studies and the expanded study, we empirically demonstrate that common structural-coverage criteria are *inadequate* when used as test generation targets. Our research identifies several caveats in the use of these coverage-criteria in the context of avionics systems. The problem is that the coverage criteria are simply too weak, which allows for the construction of ineffective tests. The auto-generated structural tests seem to be very sensitive to the test oracle chosen and the program structure of the case examples we used in our experiments. We see several possible solutions. We could potentially improve– or replace – existing structural coverage criteria to take advantage of improvements in automated test generation without allowing the generation of inefficient tests. One could also look into improvements in automated test tools or other test generation mechanisms to create longer test cases increasing the chances that a corrupted internal state would propagate to an observable output. (Chapter 4.)

**Impact of Test-suite Reduction on Test Effectiveness:** We empirically demonstrate that test-suite reduction adversely impacts the fault finding ability of test-suites that are derived from synchronous data-flow models. Even though the reduction in fault-detection effectiveness is small for some of the more rigorous test-coverage criteria such as MC/DC, we do not advocate the use of test-suite reduction techniques in this domain because of the safety-critical nature of these systems. These results also hint at advocating research into test-case prioritization techniques and experimental studies to determine if such techniques can more reliably lessen the burden of the testing effort by running a subset of an ordered test-suite, as opposed to a reduced test-suite, without little if any loss in fault finding capability (Chapter 6.)
Foundational Work for Further Empirical Research on Test Effectiveness:

Our empirical research has laid the foundation for further research in advancing our understanding of other factors that influence test effectiveness in the safety-critical systems domain. In particular, it has resulted in investigations demonstrating the importance of considering factors such as choice of test oracle, program structure, and test generation methodology for determining test effectiveness, as well as providing evidence supporting the need for us to view the whole testing process holistically (Chapter 5.)

These contributions make a positive impact on software development for safety-critical systems in several ways. First, they provide a knowledge of whether test-coverage criteria are suitable as test-generation targets in the domain of safety-critical systems. Second, the results from the empirical studies identify major caveats in the use of test automation in this domain and identify further opportunities for improving test effectiveness in the context of specification-based testing. The results from our studies has provided the foundation for further empirical research in this domain and has paved the way for investigations into the influence of multiple test artifacts on the effectiveness of specification-based structural tests.

1.6 Organization

This dissertation is organized as follows. In Chapters 2 and 3, we outline the literature review and related work. In Chapter 4, we outline how our study on test effectiveness was conducted and discuss our key observations from the study. In Chapter 5, we propose to address the threats to validity from our earlier studies through more controlled empirical studies using industrial-strength case examples; results from these studies are discussed with respect to implications for testing in the specification-based testing domain. We explore the impact of test reduction on fault
finding ability of test sets generated to various test-coverage criteria in Chapter 6.

We conclude our dissertation in Chapter 7.
Chapter 2

Background and Related Work

Our research objective is aimed at improving the quality of testing—as well as testing research—in the domain of specification-based testing. We will focus our review of related work to three areas that are most relevant to our investigation: (1) structural test-coverage criteria, (2) empirical research on test effectiveness, and (3) empirical research on the association between test reduction and test effectiveness.

2.1 Software Test Coverage (or Adequacy) Criteria

Objective measurement of test set quality is one of the key issues in software testing and has been a major research area for the last three decades. Since exhaustive testing is infeasible in practice, we can only sample test cases. A test coverage criterion describes a finite subset of test cases out of a vast or infinite number of possible tests we should execute. In other words, a test coverage criterion can be seen as a finite set of test requirements that a test suite should fulfill. We can measure coverage on any artifact produced during software development, e.g., structural coverage of source code, coverage of requirements, coverage of input space etc. A multitude of software test-coverage criteria have been proposed and studied in the literature.

In general, there are three purposes that Test Coverage Criteria serve:

1. As Stopping rules, that determine whether sufficient testing has been done.

2. As Indicators of test quality, where a degree of adequacy is associated with
each test set. In practice, the percentage of criteria coverage is often used as an criteria measurement.

3. As Test Data Generators, where a test-coverage criterion specifies a particular software testing requirement (or test generation target).

In this dissertation we are particularity interested in structural coverage criteria since they are the most commonly used in practice and mandated in some software development standards.

2.2 Structural Test Coverage Criteria

This section is devoted to test-coverage criteria for structural testing. For convenience of discussion, we classify this into two sections: one for code-based criteria and the other for specification-based criteria.

2.2.1 Code-Based Structural Test Coverage Criteria

There are two main classes of code-based structural test-coverage criteria: control-flow criteria and data-flow criteria [133, 102, 12]. Both these classes of criteria are based on the flow-graph model of program structure. Control-flow criteria focus on paths that cover control structures in the program. For instance, the decision points that determine the branch and loop structure of the program. The aim of these criteria is testing the decisions (the program points at which the control flow can divide into various paths) and conditions (atomic predicates which form the component parts of decisions) in a program. In the testing of control-flow criteria, the concepts of “decision” and “condition” need to be clearly defined. Although the definitions are not universally agreed upon in the literature, we will for the discussions in this dissertation use the definition of decisions and conditions adopted in the definition of
the Modified Decision and Condition Coverage criterion [26, 28] since this is a criterion we will use extensively in our evaluation. In this definition, a decision is a Boolean expression consisting of one or more conditions combined by logical connectives. Note here that in this definition a decision does not have to be part of a description point in a program (for example, an IF statement of loop construct), it can be part of, for example, an assignment of a complex Boolean expression to a temporary Boolean variable. A condition is an elementary Boolean expression (atomic predicate) that contains no Boolean operators.

Following are the control-flow based test-coverage criteria proposed in the testing literature [12, 126, 74, 51]. Note that these criteria are all defined over a flow graph (directed graph) representation of a program.

**Statement Coverage:** This coverage criterion requires each executable statement in the program to be covered at least once. Also known as: line coverage, segment coverage, and basic block coverage.

**Branch Coverage:** This coverage criterion requires that each boolean expression (or decision) that is part of a control structure (such as an if-statement or while-statement), evaluates to both true and false.

**Condition Coverage:** Condition coverage requires each condition that is part of a decision to evaluate both to true and false outcomes at least once. Nevertheless, full condition coverage does not guarantee full branch coverage since it may be possible to make all conditions both true and false while never changing the truth value of the decision which they are part of.

**Condition/Branch Coverage:** This coverage criterion requires each condition that is part of a decision to evaluate both to true and false at least once with the
additional requirement that the decision also evaluates to both true and false at least once.

**Multiple Condition/Decision Coverage (MC/DC):** This coverage criterion requires that every condition in a decision in the program has taken on all possible outcomes at least once, and each condition has been shown to be able to independently affect the decision’s outcome. A condition is shown to independently affect a decision’s outcome by varying just that condition while holding fixed all other conditions.

**Multiple Condition Coverage:** This coverage criterion requires that every possible combination of truth values for conditions in a decision be exercised. This coverage criterion has the problem of exponential complexity and quickly becomes infeasible when the number of conditions in a decision is large.

**All Paths Coverage:** This measure reports whether each of the possible paths in each function have been followed. A path is a unique sequence of branches from the function entry to the exit. Since loops introduce a possibly unbounded number of paths, for practical application, this measure considers only a limited number of looping possibilities.

Data-flow criteria, on the other hand, are based on the investigation of the ways in which values are associated with variables and how these associations are used in a program (the points where variables are defined and used). There are several criteria proposed in the testing literature that exercise these definition-use associations [44, 34]. A sample is included below:

**Single def-use association:** A test set that satisfies the single def-use criterion requires that for any definition occurrence of the variable at least one use occurrence is reached or covered.
**All-definitions:** A test set that satisfies the all-definitions criterion requires that it cover all definition occurrences in the sense that, for each definition occurrence, the testing paths should cover a path through which the definition reaches a use of the definition.

**All-uses:** A test set that satisfies the all-uses criterion requires that when a definition occurrence of a variable may reach more than one use occurrence, the all-uses criterion requires that all of the uses should be exercised by testing. Obviously, this requirement is stronger than the all-definition criterion.

One could extend these criteria further by distinguishing between the types of uses: either C-uses (computation use of the variable) or P-uses (predicate use of the variable). For instance, in the statement \( x := x + 2 \), the usage of \( x \) is C-use. P-Use applies to the usage of variables in conditions or branches, for instance, in the statement \( if(x > 2) \), the usage of \( x \) is P-Use. This classification of uses gives rise to the following criteria, first proposed by Rapps and Weyuker [45]:

**All P-Uses/Some C Uses:** The All-P-Uses/Some-C-Uses criterion requires that the test set exercise at least one definition-clear subpath from each definition to each predicate use reached by that definition and each successor of that use; if the definition reaches only computation uses, the test set must exercise at least one definition-clear subpath from the definition to a computation use.

**All C-Uses/Some P-Uses:** The All-C-Uses/Some-P-Uses criterion requires that a path set contain at least one definition-clear subpath from each definition to each computation use reached by that definition; if the definition reaches only predicate uses, the path set must contain at least one definition-clear subpath from the definition to a predicate use.
All P-Uses: The All-P-Uses criterion requires that the test set exercise at least one definition-clear subpath from each definition to each predicate use reached by that definition and each successor of the use.

All C-Uses: The All-C-Uses criterion requires that the test set exercise at least one definition-clear subpath from each definition to each computation use reached by that definition and each successor of the use.

Figure 2.1 shows the subsumption hierarchy amongst both the control and data flow criteria discussed in this section. The subsumption hierarchy basically suggests a partial order of data or control flow criteria; a criterion $A$ subsumes a criterion $B$ if providing 100% coverage of criterion $A$ implies that we also have 100% coverage of $B$. However, as discussed in [45], even if $A$ subsumes $B$, it is uncertain whether or not $A$ is in fact better than $B$ with respect to actually finding faults in a program under test. This is discussed further in [127], where it is shown that subsumption does not necessarily guarantee superior fault detection abilities.

This section discussed in brief the various test-coverage criteria that have been proposed in the testing literature over a flow representation of a program. In the next section, we focus instead on the various types of structural test-coverage criteria that have been defined over formal specifications.

2.2.2 Specification based Test Coverage Criteria

Specification-based structure coverage criteria specify testing requirements and measure test-adequacy according to the extent that the test-data cover the required functions specified in formal specifications. These criteria focus on the specification and ignore the program that implements the specification.

Several researchers have proposed structural test-coverage criteria defined over formal specifications of software. These are similar to traditional code-level structural
criteria discussed in the previous section, except that these are defined over a formal specification that captures the behavior of the system, as opposed to a control flow graph representation (CFG) of the program. For instance, one could define test-
coverage criteria based on the structure of the formal specification, for instance, decision coverage – which requires each guard condition to evaluate to true and false respectively. This is analogous to branch coverage criteria defined for code. One could also define data-flow-based test-coverage criteria such as All-Defs, which is satisfied iff for each variable $var$ and each defining transition, say, $t_d$ of $var$, at least one pair of $t_d$ and any using transition $t_{du}$ of $var$ is tested. This is analogous to du (def-use) pair coverage defined over a CFG representation of a program.

The following paragraphs survey some of the work done in formally defining various test-coverage criteria on formal specifications of software. Note that we are focusing on formal specifications that have been modeled as a finite state machine.

Offutt, et al. [96] define several structural test-coverage criteria that can be used to derive test inputs from state-based specifications. The criteria they define are based on traditional structural coverage since they operate over the structure of the specification and the boolean expressions guarding a state transition. Note in these definitions that a condition (or a clause) is a boolean expression that contains no boolean operators and a decision (or a predicate) is a boolean expression that is composed of conditions and zero or more boolean operators. The coverage criteria they define are the following:

**Transition coverage criterion:** This is analogous to branch coverage of code. A test set satisfying this coverage criterion must exercise every transition in the state machine.

**Transition-Pair coverage criterion:** This coverage criterion is stronger than transition coverage and requires sequences of state transitions to be exercised. A test set satisfying this coverage criterion must exercise every combination of transitions into and out of a state. This can be used to check the interfaces among the states.
**Full predicate coverage criterion:** This coverage criterion is analogous to the code-based Multiple Condition/Decision Coverage (MC/DC), which requires that every decision and every condition within the decision has taken every outcome at least once, and every condition has been shown to independently affect its decision. In the context of a state-machine based specification, where predicates are associated with state transitions, full predicate coverage requires that each decision on every transition is directly correlated to the value of each condition occurring in the corresponding decision. This is basically saying that each condition be tested independently, that is, while not being influenced by other conditions occurring within the decision.

**Complete sequence criterion:** A complete sequence is defined as a sequence of state transitions that form a practical use of the system. This requires domain knowledge expertise from the test engineer and this is not a practical test coverage criterion since the number of complete sequences is generally infinite.

Ammann and Black, et al. [6] developed a specification based **mutation test-coverage** criterion that requires test sets to distinguish between the original specification and a slightly different one (a “mutant”). The test sets are generated for mutation adequacy with respect to a state machine description such as might be found in SMV [90] or SPIN [70]. A mutant version of a specification contains a specific small syntactic error, and a test-suite is said to kill the mutant if the test-suite gives a different result over the faulty version than it does over the correct version. They also define a criterion, **UnCorrelated Full Predicate Coverage**, that is closely related to the popular multiple condition/decision coverage (MC/DC) criterion [26, 28]. This criterion is related to the Full Predicate Coverage (FP) discussed earlier, but essentially drops the requirement for decision coverage from FP. In other words, this test-coverage criterion requires tests to exercise different behaviors when
each variable in every boolean expression is true or false. They characterize this test-coverage criterion as sets of temporal logic formulas, which can then be used in conjunction with a model checker for generating coverage adequate test sets [19].

Hong, et al. [72] present a temporal logic based approach to automatic test case generation from specifications written in the Extended Finite State Machines formalism (EFSM). EFSM’s extend finite state machines with variables and operations on them and provides the underlying model for many formal specification languages such as SDL [22], Estelle [118], and Statecharts [58]. They propose several control flow and data flow oriented coverage criteria for EFSM’s and characterize them as temporal logic formulas for automatic test set generation using model checking technology. The control flow oriented criteria basically require that every state (State Coverage) or transition (Transition Coverage) in the EFSM be traversed at least once during testing. The data flow oriented criteria establishes the associations between the definition and uses of variables defined in the EFSM and requires that these associations are tested at least once during testing. They consider two types of associations: definition-use pairs and affect pairs, which are central notions in data flow analysis and program slicing, respectively [124, 73]. The affect pairs define associations between the use of a variable that directly or indirectly affects the definition of another variable in the EFSM. They propose two coverage criteria for exercising dataflow among local variables in the EFSM (The variables in the EFSM are partitioned as input, local, and output variables):

**All-def Coverage:** A test-suite satisfies this coverage criterion, if, for every definition of a variable in the EFSM, some testable du-pair is covered by a test sequence in the test-suite.

**All-use Coverage:** A test-suite satisfies this coverage criterion, if, for every definition of a variable in the EFSM, every testable du-pair is covered by a test
sequence in the test-suite.

In order to exercise data flow among input and output variables, they propose two additional criteria that involve affect pairs, i.e., pairs of variables that start with an input variable and end with an output variable. Such affect-pairs are defined as input-output pairs.

**All-input Coverage:** A test-suite satisfies this coverage criterion if, for every use of every input variable in the EFSM, some testable input-output pair is covered by a test sequence in the test-suite.

**All-output Coverage:** A test-suite satisfies this coverage criterion if, for every use of every input variable in the EFSM, every testable input-output pair is covered by a test sequence in the test-suite.

Rayadurgam, *et al.* [106] provide a generic formalism for structural test case generation using model checkers and illustrate how this approach can be applied to the formal specification language (RSML-ε). RSML-ε (Requirements State Machine Language without events) is a state-based specification language based on hierarchical finite state machines and dataflow languages. The language supports parallelism, hierarchies, and guarded transitions. An RSML-ε specification consists of a collection of input variables, state variables, input/output interfaces, functions, macros, and constants; input variables are used to record the values observed in the environment, state variables are organized in a hierarchical fashion and are used to model various states of the control model, interfaces act as communication gateways to the external environment, and functions and macros encapsulate computations providing increased readability and ease of use. Transitions between the states are guarded by complex boolean expressions (expressed either as simple conditions or in Disjunctive Normal
The language uses a tabular notation called AND/OR Tables to express complex boolean expressions in Disjunctive Normal Form. This was adopted from the original RSML notation, where each column of truth values represents a conjunction of the propositions in the leftmost column (a “*” represents a “don’t care” condition). If a table contains several columns, we take the disjunction of the columns. They define a set of control-flow oriented test-coverage criteria for RSML-e. These criteria are similar to traditional code based structural test coverage criteria, and are intended to provide adequate coverage on the model expressed in this formalism. Following are the test coverage criteria they define:

**State Coverage**: A test-suite is said to achieve state coverage of a state variable in an RSML-e specification, if for each possible value of the state variable there is at least one test case in the test-suite that assigns that value to the given variable. The test-suite achieves state coverage of the specification if it achieves state coverage for each state variable.

**Decision Coverage (Table Coverage)**: A test-suite is said to achieve decision coverage of a given RSML-e specification, if each guard condition (specified as either an AND/OR table or as a standard Boolean expression) evaluates to true at some point in some test case and evaluates to false at some point in some other test case in the test-suite.

**Clause-wise transition coverage**: This test coverage criterion is analogous to modified condition/decision coverage (MC/DC). A test-suite is said to achieve clause-wise transition coverage (CTC) for a given transition of a variable in an RSML-e specification, if every basic Boolean condition in the transition guard is shown to independently affect the transition.

This dissertation work focusses primarily on the use of specification-based struc-
tural test-coverage criteria for purposes of test-data generation using counterexample based techniques proposed in Chapter 1.

2.3 Empirical Studies on Test Effectiveness

Various comparisons have been made of the theoretical effectiveness various test-coverage criteria [34]. However, the questions of real concern to researchers and potential users of these criteria deal with their actual effectiveness in detecting the presence of faults in programs. Test practitioners would like to know whether the effort they spend in using these criteria is well spent in terms of the cost in generating test to satisfy the criteria and the potentially payoff in terms of increased fault detection capability of the resultant test sets. In an effort to answer this question, several studies have been conducted by researchers in academia and industry. This section highlights the important studies performed in this area.

Frankl, et al. [44] report on an empirical evaluation of the fault detecting ability of two-white box software testing techniques: decision coverage (branch testing) and the all-uses data flow testing criterion. The subjects used here were eight versions of an antenna configuration program written for the European Space Agency, each consisting of over 10,000 lines of C Code. Each subject program was tested using a very large number of randomly generated test sets. For each test set, the extent to which it satisfied the given testing criterion was measured and it was determined whether or not the test set detected a program fault. These data were used to explore the relationship between coverage achieved by test sets and the likelihood that they will uncover a fault. For each of the subject programs considered, the likelihood of test set detecting a fault increased as high coverage levels were reached. Also, for all of the subject programs considered, test sets that attained a high level of decision coverage or definition-use association were significantly more likely to detect
the fault than random test sets of the same size. The data they obtain supports
the belief that these structural testing techniques can be more effective than random
testing. However, the studies were performed on a small sample of the population of
possible programs and they still leave open the question as to whether the benefits
of these techniques outweigh the costs.

Hutchins, et al. [75] report on an experimental study investigating the effective-
ness of two code-based test-adequacy criteria for identifying sets of test cases that
detect faults. The all edges and all-DUs (modified all-uses) coverage criteria were ap-
plied to 130 faulty program versions derived from seven moderate sized programs by
seeding realistic faults. Several thousand test sets for each faulty program were gen-
erated and examined the relationship between fault detection and coverage. Within
the limited domain of their experiments, test sets achieving coverage levels over 90
percent usually showed significantly better fault detection than randomly chosen test
sets of the same size. In addition, significant improvements in the effectiveness of
coverage-based tests usually occurred as coverage increased. However, the results
also indicate that code coverage alone is not a reliable indicator of the effectiveness
of a test set. The findings in this study are similar to those found in the Frankl, et
al. studies mentioned above.

A study by Basili and Selby [10] attempts to compare three techniques: code
reading by stepwise abstraction, functional testing using equivalence partitioning and
boundary value analysis, and structural testing using 100 percent statement coverage.
The study evaluates the techniques across three data sets in three different aspects of
software testing: fault detection effectiveness, fault detection cost, and classes of faults
detected. The three data sets involved a total of 74 programmers applying each of the
three testing techniques on unit-sized software. The unit-sized programs contained
a total of 36 faults in a fractional factorial experiment, giving observations from 222
testing sessions on the effectiveness of the testing methods. The programs considered were: a text formatting program, a mathematical plotting routine, a numeric data abstractor consisting of a set of list processing utilities, and a maintainer program for a database of bibliographic references. The faults contained in the programs tested represent a reasonable distribution of faults that commonly occur in software. The major empirical results from this study are the following: (1) Code reading (with professional programmers) detected more faults than did functional or structural testing, while functional testing detected more faults than structural testing, (2) Number of faults observed, fault detection rate, and total effort in detection depended on the type of software tested. (3) Functional testing detected more control faults than did other methods. The results suggest that code reading by stepwise abstraction (a non execution-based method) is at least as effective as functional and structural testing in terms of number and cost of faults observed. The comparison with structural testing is rather inadequate since they focus on statement coverage, which is rather a weak coverage criterion. Their results regarding the ineffectiveness of statement coverage is something that is expected. One would have liked to see a more broader variety of structural test-coverage criteria evaluated in these experiments.

Offut, et al. perform an experimental evaluation of dataflow and mutation testing. The dataflow criteria they use is DU pair coverage and the mutation test criterion requires tests to distinguish between the original specification and a slightly different one (“a mutant”). They compare these techniques by evaluating the effectiveness of test-data developed for each. For this purpose, they used a set of 10 program units written in C for conducting their experiments. For each program, they generated test sets that were mutation adequate and test sets that were dataflow adequate. To avoid any bias, they generated five independent test sets for each criterion. Thus, for each program, they had ten test sets; five mutation-adequate and five dataflow-
adequate, amounting to a total of 100 test sets. They then create faulty programs by introducing the following types of faults: deleting conditional or iterative clauses, changing conditional expressions by adding extra conditions, changing initial values and stop conditions of iteration variables etc. They run the coverage adequate test sets for each criterion over the faulty programs and determine if they exposed the fault. The results of their experiment seemed to indicate that although mutation adequate and data flow adequate test sets detected a significant percentage of the faults, the mutation adequate test sets detected an average 16% more faults than the data flow adequate test sets. However, the programs they studied were relatively small and not "representative" (since it is difficult to choose a representative sample of software, test cases, or faults) and therefore cannot be generalized to larger software systems.

Foreman and Zwebens study [43] examined the effectiveness of several variants of controlflow and dataflow criteria in detecting thirty of the real faults that were documented by Knuth during development of the program. A testing criterion was considered effective at detecting a fault only when all test sets satisfying the criterion revealed the fault. Application of the all-uses criterion guaranteed detection of thirteen of the thirty faults. Eleven of these thirteen were also guaranteed to be detected by tests satisfying branch coverage. It is shown that even the simplest of the control and data flow strategies are effective at revealing a reasonable number of the defects. For those defects for which the control and data flow strategies were not effective, other well-known methods such as special value testing, boundary testing, or static data flow analysis were usually effective. The results suggest that systematic testing strategies for which tool support is possible can eliminate a significant number of defects undetected by current testing practice.

A study by Thevenod-Fosse, Waeselynck, and Crouzet [119] investigates the fault
revealing powers of different test patterns derived from ten structural test criteria. The ten criteria under investigation were path selection criteria and include three criteria related to control flow: statement coverage, branch coverage, and all paths coverage. The seven other criteria are based on data-flow analysis: All-DU Paths, All-Uses, All-P-Uses/SomeC-Uses, All-C-Uses/SomeP-Uses, All-P-Uses, All-C-Uses, All-Defs. The experiments were performed on four programs written in C, from a nuclear control plant system. Three test input generation techniques were studied: 1. based on deterministic choice 2. random selection based on an input probability distribution (called structural statistical testing), 3. random selection from a uniform distribution on the input domain. Mutation analysis was used to determine the test set fault detection ability and mutation scores were used to compare the effectiveness of deterministic structural testing techniques to their own method of test generation (structural statistical testing). The mutations were automatically created the four small C programs, creating a total of 2914 mutants; equivalent mutants were eliminated by hand. The test sets for the deterministic part of the testing were created by hand for each criterion under consideration; for each criterion and for each program, at most 10 test sets were designed, with at most 19 members in each test set. The resulting mutation scores of these test sets were used to determine the relative effectiveness of the methods. The study concluded that structural statistical testing was more effective at killing mutants than the other two test generation strategies.

A more recent study was conducted by Fraser and Zeller [47] where they evaluate test suites generated using mutation testing with manually written test suites. Using mutation analysis techniques, the test inputs are first generated and then assertions capable of distinguishing the mutants from the program, with respect to the test inputs, are created. They use mutations rather than structural properties as coverage criteria to get guidance on where to test, but also what to test for. They conducted
their investigations using case examples from 10 projects containing a few hundred classes written in the Java programming language. In their experiments, they report that test suites generated using mutation analysis are better at finding defects than manually written test suites. However, they do not compare against automatically derived tests for various structural coverage criteria and their evaluation results are not generalizable since they are based on a few case examples. Their evaluation is seen as investigating the potential of their technique rather than providing a general statement on test effectiveness.

The previous paragraphs discussed various studies performed on measuring test effectiveness in the traditional program-based testing domain. To our knowledge, in the formal specification-based testing domain there has been just one small case study on evaluating various test-coverage criteria. The case study was performed by Offutt et al. [97], where they make an experimental comparison of data flow adequate and mutation adequate test-data tests in terms of their ability to find faults. A small sized program was used (Cruise Control), representative faults were seeded by hand, and tests manually generated for the two test-adequacy measures. Their results seem to indicate that both these criteria were effective in finding faults in the subject programs that were used for the study, with the mutation adequate sets performing better. However, the subject programs that they studied were relatively small, and the coverage criteria considered in the study were rather weak, thereby leaving the question open as to how their conclusions might scale up to large software systems.

Clearly, there is a need for more experimentation in this specification-based testing domain to understand the usefulness and relationships amongst various structural test-coverage criteria defined over these specifications. The experiments we wish to perform on test effectiveness would address a wider range of formal specification-based test-coverage criteria ranging from the weakest (like state coverage) to the strongest
(like MC/DC). Our main purpose in the experiments is to evaluate coverage adequate test sets in terms of their ability to detect faults. Most of the earlier studies in the program-based testing domain study primarily the effect of coverage levels on the fault detection effectiveness. Also, we propose to use several formal specifications available to us which are larger and closer to realistic sized systems in industry.

2.4 Empirical Studies on association between Test Reduction and Test Effectiveness

In the traditional program based testing domain, two major studies have investigated the effect of test-set reduction on the size and fault finding capability of a test-set. In the first study, Wong et al. address the question of the effect on fault detection of reducing the size of a test set while holding coverage constant [129, 130]. Their experiments were carried out over a set of commonly used UNIX utilities implemented in C. These programs were manually seeded with faults, producing variant programs each of which contained a single fault. They randomly generated a large collection of test sets that achieved block and all-uses data flow coverage for each subject program. For each test set they created a minimal subset that preserved the coverage of the original set. They then compared the fault finding capability of the reduced test-set to that of the original set. Their data shows that test minimization keeping coverage constant results in little or no reduction in its fault detection effectiveness. This observation leads to the conclusion that test cases that do not contribute to additional coverage are likely to be ineffective in detecting additional faults.

In the second study, Rothermel et al. performed a similar experiment using seven sets of C programs with manually seeded faults [109]. For their experiment they used edge-coverage [45] adequate test-suites containing redundant tests and compared the
fault finding of the reduced sets to the full test sets. In this experiment, they found that (1) the fault-finding capability was significantly compromised when the test-sets were reduced and (2) there was little correlation between test-set size and fault finding capability. The results of the Rothermel study were also observed by Jones and Harrold in a similar experiment [81].

The results of these two studies are in stark contrast to each other and this can be attributed to several potential factors: (1) The subject programs utilized were different, (2) Different mechanisms were used to generate reduced test sets, (3) test-suites were minimized for different test criteria. The Wong study uses all-uses coverage, whereas the Rothermel study uses test sets that satisfy all-uses coverage criteria. Finding the association between coverage criteria, test-suite size, and fault finding capability requires a controlled comparative study, something that is not fully addressed in both these studies.

To our knowledge there has been no study performed on assessing the correlation between test set reduction and fault detection effectiveness of test sets that are derived from formal specifications (using various structural test-coverage criteria) expressed in synchronous data-flow languages commonly used in specification-based development, for example, Esterel, SCADE, SpecTRM, SCR, and RSML. As in the experiments described in this dissertation for measuring test effectiveness, we are seeking to address a wide variety of coverage criteria ranging from the very weak, for example, transition coverage, to the very strong, for example MC/DC. The previous experiments addressed either rather weak criteria such as block-coverage [130] or used test-suites that did not provide complete coverage [81].
2.5 Chapter Conclusion

This chapter discussed the notion of test-coverage criteria and provided a brief summary highlighting the different types of coverage criteria that have been proposed in the testing literature in both the traditional program based testing domain and more recently, in the specification-based testing domain.

Relevant empirical research on the effectiveness of various structural test-coverage criteria (both in the code based testing domain and specification-based testing domain) were highlighted and discussed. In Chapters 4 and 6 of this dissertation, we will compare and contrast these previous studies with our own empirical research.

In the next chapter we will discuss how these structural test-coverage criteria can be used in the context of specification-based testing to automatically generate test data using a test generation mechanism known as *model checking*. 
Chapter 3

Automated Test Case Generation from Formal Models of Software

Specification-based testing advocates have long promoted the use of formal behavioral models as a source for test generation and test oracles. Our work focuses on a class of systems known as reactive systems, which continually interact with their environment. The behavior of such systems can be described using formal specification languages which are discussed in the following sections. Once such a description is available, it lends itself to be used in conjunction with test generation mechanisms such as model checking. Although there are several ways to generate test data, we will restrict the scope of our work to model checkers since they have proven to be able to effectively generate test cases for the types of systems we are studying [63].

3.1 Formal Specification of Synchronous Reactive Systems

Reactive systems are systems whose role is to maintain an ongoing interaction with their environment rather than produce some final value upon termination. Typical examples of reactive systems are Air traffic control system, Programs controlling mechanical devices such as a train, a plane, or ongoing processes such as a nuclear reactor. The behavior of such systems can be specified using formal specification languages.

Formal specification languages are a way of specifying and verifying such systems
by applying techniques from mathematics and logic. This enables the analysis of systems and the reasoning about them with mathematical precision and rigor. In particular, a formal, and formally verified specification provides a more precise, more consistent and more complete starting point for the testing process and the obtained tests can be formally validated whether they test what should be tested. Moreover, the use of formal methods allows automating the generation of tests from formal specifications, thus leading to a faster, cheaper and less error-prone testing process.

Formal specification languages differ mainly by the particular paradigm they rely on. For convenience, we can classify them into the following categories:

**System-History-based specifications:** The principle here is to specify a system by characterizing its maximal set of admissible histories (or “behaviors”) over time. The properties of interest are specified by temporal logic assertions about system objects; such assertions involve operators referring to past, current and future states. The assertions are interpreted over time structures. Time can be linear [103]or branching [39]. Time structures can be discrete, dense, or continuous. The properties may refer to time points, time intervals, or both [38]. Most often it is necessary to specify properties over time bounds; real-time temporal logics are therefore necessary [69].

**Model-based specifications:** Instead of characterizing the admissible system histories, one may characterize the admissible system states at some arbitrary snapshot. The properties of interest are specified by (a) invariants constraining the system objects at any snapshot, and (b) pre- and post-assertions constraining the application of system operations at any snapshot. A pre-assertion captures a weakest necessary condition on input states for the operation to be applied; a post-assertion captures a strongest effect condition on output states if the operation is applied. The latter may be explicit or implicit dependent on
whether or not the assertion contains equations defining the output constructively. Languages such as Z [11], VDM [42, 52] rely on this paradigm.

**Transition-based system specifications:** Instead of characterizing admissible system histories or system states, one may characterize the required transitions from state to state. The properties of interest are specified by a set of transition functions in the state machine transition; the transition function for a system object gives, for each input state and triggering event, the corresponding output state. The occurrence of a triggering event is a sufficient condition for the corresponding transition to take place (unlike a precondition, it captures an obligation); necessary preconditions may also be specified to guard the transition. Languages such as SCR [17], StateCharts [58], Estelle [118], SDL [22] etc. are based on this paradigm.

**Algebraic Functional specifications:** The principle here is to specify a system as a structured collection of mathematical functions. The functions are grouped by object types that appear in their domain or codomain, thereby defining algebraic structures (or abstract data types). The properties of interest are then specified as conditional equations that capture the effect of composing functions (typically, compositions with type generators). Languages such as ASL [35] rely on this paradigm.

**Higher-Order Functions:** Here too the system is specified as a structured collection of mathematical functions, but the functions themselves are grouped into logical theories. Such theories contain type definitions (possibly by means of logical predicates), variable declarations, and axioms defining the various functions in the theory. Functions may have other functions as arguments which significantly increases the power of the language. Languages such as HOL [5]
or PVS [114, 98, 99] rely on this paradigm.

In this dissertation, we will be focussing on synchronous dataflow languages that are most suited to expressing the behavior of safety-critical reactive systems. The next section discusses these class of languages with respect to the classification scheme discussed here.

3.2 Synchronous Dataflow Languages

Synchronous dataflow languages [104, 14, 55, 54] were proposed as a software engineering tool in the late 1970s and independently in the programming language community in the late 1980s as a technique for modeling and constructing reactive (i.e., process control) systems. Synchronous dataflow languages provide primitives that allow programmers to naturally model the dataflow, and also to consider that their programs as reacting instantaneously to external events. Statecharts [58], RSML [83], RSML$^{-e}$ (RSML without events) [65, 122, 123], Lustre [54], SCR [67] fall under the category of synchronous data flow languages. Researchers have explored approaches to test-case generation from specifications of software systems expressed in this class of languages. These languages have their associated toolsets for prototyping, simulation and verification. For instance, NIMBUS [120, 121, 123], is the toolset that supports the RSML$^{-e}$ formalism. Esterel/SCADE [15] and Matlab/Simulink [87] are toolsets with similar capability.

Once a formal specification is available, it lends itself to automation for a variety of purposes such as code generation, test case generation. For our purposes, we are primarily interested in a mechanism for automating the generation of test cases from such a formal specification in the context of specification-based testing, which we will discuss below.
3.3 Specification-Based Testing

Specification-Based testing [127, 21, 92, 20] relies on the use of a formal specification to drive test sequence generation. This approach is cost effective because it can be used to automate the generation of test cases and makes it easier to handle requirements evolution by modifying the model and regenerating tests, rather than just maintaining the test-suite. This can dramatically reduce the cost of test maintenance.

As briefly mentioned in the introduction section of this dissertation, the specification-based testing process can be conveniently divided into the following five main steps as shown in Figure 3.1. In this section, we will go through each of these steps in
greater detail.

1. Create an Abstract Model of the program or system under test (SUT) and/or its environment. Specification-based testing relies on the creation of an abstract model because it is much smaller and simpler than the SUT itself. The abstract model will focus on key aspects that we want to test and removes many of the lower level details of the SUT under test. While creating this model, we may also want to annotate it to clearly document the relationship between the informal requirements and the formal model that is created from it. In this thesis, we choose a specification language which has formal semantics, such as a semantics based on Finite State Machines (FSM). The semantics of an FSM based language describes how a specification written in that language can be translated automatically into a corresponding FSM that describes the behavior of the specified system. Automatic test generation assumes that the implementation, derived from such a specification, has the same properties as the FSM, for example, states and state transitions. The algorithms for automatic test generation take advantage of this assumption by exercising the FSM and searching for test sequences that cover states and state transitions of the FSM. Most formal notations provide some automated verification tools such as type checkers and static analysis tools that allow us to explore the behavior of the specification and check the expected behavior.

2. Generate abstract tests from the specification. We choose test selection criteria, to say which tests we want to generate from the formal specification, because there are usually an infinite number of possible tests. For example, we might interact with the test generation tool to choose a particular specification-based coverage criterion, such as transition coverage, or we might focus on a particular portion of the specification. The use of specification-based coverage criteria is
key to the automation process as it specifies the test obligations and drives our test generation mechanism to choose the test inputs that will satisfy the particular test obligation. The tests also include the expected output values. The main output of this step is a set of abstract test cases, which are sequences of operations over the specification. These abstract tests lack some of the detail needed by the SUT and are not directly executable on it. The abstract tests, however, can be executed on a simulation environment for the specification language to observe the expected behavior of the system.

3. **Concretize** the abstract tests to make them executable. The third step of specification-based testing is to transform the abstract tests created in step 2 into executable concrete tests that can be run on the SUT. This may be done by a *transformation tool*, which uses mappings to translate each abstract test case into an executable test script. Or it may be done by writing some *adaptor* code that wraps around the SUT and implements each abstract operation on terms of lower-level SUT services. The overall goal here is to add low-level SUT details to the concrete test cases that were not present in the abstract model of the software system. One advantage of having abstract tests is that it is independent of the language used by the test execution environment of the SUT. We can reuse the same abstract tests on a different test execution environment by simply modifying the adaptor code or the translation templates.

4. **Execute** the tests on the program or system under test (SUT) and obtain test results. The fourth step is to execute the concrete tests on the system under test. We can do through online means, where the tests are executed as they are produced, or through offline means, where the test scripts are executed after we generate all the concrete tests. The abstract tests are also executed on the specification using its simulation environment, thereby giving us the expected
behavior of the system. The oracle is part of the abstract test case as the expected output is generated for each test step.

5. **Analyze** the test results. The last step is to analyze the results of the test executions and take corrective action. The test oracle which is derived from the specification will tell us if the test passed or failed. For each test case that reports a failure, we need to determine the fault that caused that failure. This is very similar to the traditional test analysis process. When a test fails, we may find that the failure may be due to a fault in the SUT, or it is due to a fault in the test case itself. Since we derived the test case from the specification, this could be traced to a problem in the adaptor code or a problem in the model itself.

Researchers in specification-based testing have investigated the use of formal verification tools that use model checking technology for purposes of automating test sequence generation from these formal specifications. The next section discusses the basic approach behind model checking technology and how this can be leveraged for automatic test-data generation.

### 3.4 Model Checking

Model checking [8, 131, 91, 18, 31, 3, 80] is an automatic technique for verifying finite-state reactive systems, such as sequential circuit designs and communication protocols. This technique has several important advantages over mechanical theorem provers or proof checkers for verification of finite-state reactive systems. The most important advantage is that the procedure is automatic. Typically, the user provides a high level representation of the model and the property to be checked. The model checker will either terminate with the answer *true*, indicating that the model satisfies
the property, or give a counterexample execution showing why the property is not satisfied.

The properties the model checker is checking for are cast as formulas in an appropriate temporal logic and the system behavior is specified as some form of a transition system. When a model checker detects a property violation, it will produce a counterexample, which is a sequence of inputs that will take the finite state model from its initial state to a state where the violation occurs. Many temporal logics like CTL (Computation Tree Logic) [39] and LTL (Linear Time Temporal Logic) [103] are popular ways of specifying the desired properties of interest in the system. These come with their associated model checking systems such as the explicit-state model checker SPIN [70], the symbolic model checker NuSMV [91], and bounded model checkers such as SAL [13, 36]. A thorough discussion of temporal logics is beyond the scope of this dissertation, but we will provide a brief overview of CTL and LTL temporal logics, when used in model checking context, in the following sections.

3.4.1 Temporal Logics for Model Checking

Model checking as briefly discussed above, offers an attractive means of making queries about state transition systems. In model checking, one describes a property of the transition system in a temporal logic and then invokes a decision procedure for traversing the state transition graph of the system and determining whether the property holds for that system. The exact procedure will vary with the temporal logic and the type of formula; furthermore, for a particular logic and a particular type of formula, there may be several, equally sound model checking procedures. Temporal logic is an extension of classical logic. CTL and LTL are two common ways of specifying desired properties of interest in a system. The following sections provide a brief discussion on these types of temporal logics.
3.4.2 Computation Tree Logic

Computation Tree Logic (CTL) is defined over Kripke Structures [84], which are structures having a set of states, a subset of these being initial states, and a transition relation. In addition, Kripke structures have what is known as a labeling function, which assigns to states certain atomic propositions from an underlying set of atomic propositions.

CTL is built from path quantifiers and temporal operators. There are two path quantifiers, $A$ and $E$, with meanings as follows:

- $A$, "for every path"
- $E$, "there exists a path"

A path is an infinite sequence of states such that each state and its successor are related by the transition relation. CTL has four temporal operators. We give English language interpretations of these, below, and for convenience, we assume $p$ and $q$ are purely propositional formulas:

- $Xp$, "$p$ holds at the next time step"
- $Fp$, "$p$ holds at some time step in the future"
- $Gp$, "$p$ holds at every time step in the future"
- $pUq$, "$p$ holds until $q$ holds"

A time step is usually identified with a computation step, such as, a clock tick in synchronous systems. The syntax of CTL dictates that each usage of a temporal operator must be preceded by a path quantifier. These path quantifiers and temporal operator pairs can be nested arbitrarily and the CTL formulae may also be connected by Boolean operators, for instance, $EF(p \land q)$, $AG(r \rightarrow x)$ etc.

The model checking problem for CTL is, then, to take a formula such as those mentioned above, and determine whether the set of states satisfying the formula in a particular system include the initial states of the system. If this is the case, then it
is said that the system satisfies the formula.

### 3.4.3 Linear Time Temporal Logic

In Linear Time Temporal Logic (LTL), we can encode formulae about the future of computation paths, e.g., a condition will eventually be true, or a condition will always be true, or a condition will be true until another condition becomes true, etc. An LTL formula consists of a finite set of propositional variables, the logical operators ($\wedge$, $\vee$, $\sim$, $\rightarrow$, and $\leftrightarrow$), and the fundamental temporal modal operators $X$, $G$, and $F$. For instance, the formula $Xp$ specifies that property $p$ holds at the next time step. The Globally operator $G$ is used for specifying safety properties. For instance, a formula $Gp$ holds along a path if $p$ holds in all states of the path. The modal operator $F$ is used to specify liveness properties. For instance, the formula $Fp$ holds along a path if $p$ holds somewhere on the path. In this informal introduction to temporal logic, we will avoid a detailed explanation of the binary temporal operators $Until(U)$ and $Release(R)$. A good discussion on these can be found in [125].

The set of LTL formulas over a finite set of propositional variables can be inductively defined over the set of propositional formulas as follows:

- If $p \in AP$, where $AP$ is a finite set of propositional variables, then $p$ is a LTL formula;
- if $\alpha$ and $\beta$ are LTL formulas, then $\sim \alpha$, $\alpha \vee \beta$, $X\alpha$, $G\alpha$, $F\alpha$, $\alpha U \beta$ are LTL formulas

In this dissertation, we will focus on LTL, because it is a temporal logic that is well known in the area of formal verification, and has intuitive semantics that are better suited for our purposes in specifying system properties as compared to logics such as CTL. In linear-time model checking one searches for a counterexample trace, finite, or infinite, which falsifies the desired property. The standard technique
for model checking LTL is to compute the product of a Kripke structure with an automaton that represents the negation of the property (this automaton captures exactly the execution sequences that violate the LTL formula). Emptiness of the product automaton is an evidence of the correctness of the property.

Given the use of temporal logics for specifying desired properties of interest in a system, there are several strategies for exploring reachable states: (1) explicit model checking, where the model checker uses an explicit representation of states and enumerates the set of reachable states by forward exploration, (2) symbolic model checking, where a breadth first search of the state space is effected through the use of ordered binary decision diagrams (OBDDs) [24, 23]—which hold the characteristic function of sets of states, this allowing computation of transition among sets of states rather than individual states, and (3) bounded model checking, where the Finite state machine is searched for property violation using a fixed number of steps. This typically involves encoding the model as an instance of a satisfiability problem. We will discuss these three strategies in greater detail in the following sections.

3.4.4 Explicit State Model Checking

Explicit state model checking uses an explicit representation of the system’s global state graph, usually given by a state transition function. An explicit state model checker evaluates the validity of the temporal properties over the model by interpreting its global state transition graph as a Kripke structure [84], and property validation amounts to a partial or complete exploration of the state space. For example, The SPIN model checker [70] is the most prominent explicit state model checker mainly used for checking communication protocols. To perform verification, SPIN takes a correctness claim that is specified as a temporal logic formula, converts that into a Büchi automaton [33], and computes a synchronous product of this claim and the
automaton representing the global state space or the global reachability graph. The is again represented as a Büchi automaton. If the language accepted by this automaton is empty, this means that the original claim is not satisfied for the given system. If the language is nonempty, it contains precisely those behaviors that satisfy the original temporal logic formula.

However, due to the large size of the state space for realistic systems it is hardly ever possible to explore the full state space in order to decide about the correctness of the system. In such a case, it either takes too long to explore all states in order to give an answer within a useful time span, or the size of the state space is too large to be stored within the bounds of available memory.

### 3.4.5 Symbolic Model Checking

Finite state models of concurrent systems grow exponentially as the number of components of the systems increases – this is known widely as the state space explosion problem in automatic verification. To avoid this problem, a method called symbolic model checking has been proposed and studied. This method avoids building a state graph by using Boolean formulas to represent sets and relations. The state space can be traversed much more efficiently by considering a representation of state sets and defining transition relations on them. One good candidate for such a symbolic representation is the Ordered Binary Decision Diagram (OBDD) [24, 23], which is widely used in various tools for the design and analysis of digital circuits. These decision graphs provide a compact canonical form for Boolean functions. To apply this idea to temporal verification, we observe that if a state space is represented by a vector of Boolean variables, then a set of states can be represented by a Boolean function which returns true for all states in the set. Similarly, a relation $xRy$ between states can be a represented by a Boolean function of two sets of variables, one representing $x$ and
the other representing $y$. In this way, a model checking algorithm is developed which uses OBDDs to represent sets and relations. A variety of properties characterized by least and greatest fixed points can be verified purely by manipulations of these formulas using Ordered Binary Decision Diagrams. The process roughly works as follows: The set of initial states is represented as a BDD. The procedure then starts an iterative process, where at each step $i$, the set of states that can be first reached in $i$ steps from an initial state are added to the BDD. At each step, the set of new states is intersected with the set of states that satisfy the negation of the property. If the resulting set is non-empty, it means an error has been detected. This process terminates when the set of newly added states is empty or an error is found. The first case indicates that the property holds, because no reachable state contradicts it. In the latter case, the model checker produces a counterexample. Note that termination is guaranteed, since there are only finitely many states. Using symbolic model checking, we can automatically verify some regularly structured systems with literally astronomical numbers of states.

While BDD based symbolic model checking has enabled an order of magnitude increases in the size of designs that could be verified – thanks to improvements in abstraction and compositional reasoning – this is still inadequate for many industrial designs of interest.

### 3.4.6 Bounded Model Checking

In bounded model checking, we construct a Boolean formula that is satisfiable if and only if the underlying state transition system can realize a finite sequence of state transitions that reaches certain states of interest. If such a path segment cannot be found at given length, $k$, the search is continued for larger $k$. The procedure is symbolic. i.e., symbolic Boolean variables are utilized; thus, when a check is done for
a specific path segment of length \( k \), all path segments of length \( k \) are being examined. The Boolean formula that is formed is given to a satisfiability solving program and if a satisfying assignment is found, that assignment is a witness for the path segment of interest.

Essentially, there are two steps in bounded model checking. In the first step, the sequential behavior of a transition system over a finite interval is encoded as a propositional formula. In the second step, that formula is given to a propositional decision solver, i.e., a satisfiability solver, to either obtain a satisfying assignment or to prove there is none. Each satisfying assignment that is found can be decoded into a state sequence which reaches states of interest. There are several advantages of bounded model checking. SAT tools, like SATO [132], and GRASP [86], do not require exponential space and large designs can be checked fast, since the state space is searched in an arbitrary order. BDD based model checking usually operates in breadth first search consuming much more memory. Further, the procedure is able to find paths of minimal length with remarkable efficiency. Another comparative advantage is that bounded model checking seems to require little by-hand manipulation from the user, while BDD based verifications often require a suitable, manual ordering for BDD variables.

The disadvantage of bounded model checking, is that the method lacks completeness and the types of properties that can currently be checked are very limited. For instance, favorable results have been obtained for both safety and liveness properties, where the verification of a safety property involves checking whether a given set of states is reachable, and the verification of an eventuality property involves detecting loops in a system’s state transition graph.

In this dissertation, we will use bounded model checking that uses properties specified in LTL. The reasoning behind this is that, in bounded model checking, the
search is restricted to a trace of bounded length, where the bound is selected before the search – The idea being that many errors can be found in traces of relatively small length, and for the systems we are investigating, i.e., in the domain of safety-critical reactive systems, we expect the test cases to be of short length. The restriction to bounded length traces enables a reduction to propositional satisfiability and SAT based model checking has been shown to be more efficient and significantly outperforms BDD based model checking. As bounded model checking is essentially a search for counterexample traces of bounded length, it fits naturally within the linear time framework (LTL) and not within the branching time framework (CTL).

3.5 Model Checking for Automatic Test-Data Generation

As discussed above, Model checkers build a finite state transition system and exhaustively explore the reachable state space searching for violations of the properties under investigation and thus, offer an effective approach to automated test generation based on their ability to generate counterexamples to invalid assertions. Roughly speaking, to generate a test case that will exercise a behavior characterized by a predicate p, we model check for the property “always not p” and the counterexample to this property will provide the required test case that will exercise the predicate p.

model checker. For instance, we may want a test case to cover a particular state A in the formal specification. We can formulate this as a condition describing a test case covering this state i.e., the sequence of inputs must take the model to state A; This is a property expressible in the logics used in common model checkers, for example, the logic LTL (Linear Temporal Logic). We can then trap 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 [49]. The model checker will now search for a counterexample demonstrating
that this trap property is, in fact, satisfiable; the counter example thus serves as a test case that will cover the state of interest. One can thus characterize test coverage criteria as a set of temporal logic properties. The test generation approach discussed above can be used to generate test suites that satisfy a wide variety of structural test-coverage criteria defined over a formal specification.

Several research groups are actively pursuing model checking techniques as a means for test generation.

Gargantini and Heitmeyer [49] describe a method for generating test sequences from requirements specified in the SCR notation. To derive a test sequence, a trap property is defined which violates some known property of the specification. In their work, they define trap properties that exercise each case in the event and condition tables available in SCR [66] – this provides a notion of branch coverage of the SCR specification.

Amman and Black [7] combine mutation analysis with model-checking based test case generation. They define a specification-based coverage metric for test suites using the ratio of number of mutants killed by the test suite to the total number of mutants. Their test generation approach uses a model-checker to generate mutation adequate test suites. The mutants are produced by systematically applying mutation operators to both property specifications and the operational specifications, producing respectively, both positive test cases which a correct implementation should pass, and negative test cases which a correct implementation should fail.

Rayadurgam et.al. [108, 107] in provide a formalism suitable for structural test-case generation using model checkers and in illustrate how this approach can be applied to a formal specification language. They also presented a framework for specification centered testing in.

Lee, et.al. [71] formulate a theoretical framework for using temporal logic to specify
data flow test coverage criteria. They also discuss various techniques for reducing the size of the test set generated by the model checker. The underlying argument in all these works, is that when test criteria can be appropriately formulated as temporal logic formulas, one could use model-checking to produce witnesses for those formulas, which could then be seen as test sequences satisfying the coverage criteria.

Rayadurgam et.al. [63] also conducted an a series of studies using realistic systems to examine how well model checking techniques perform for test-data generation purposes, and to what extent they scale up. The experimental data point to limitations of symbolic as well as explicit state model checkers. In this experiment, the bounded model checker performed very well in the domain of safety critical systems. In this domain, test cases providing common coverage seem to be quite short, thus making bounded model checking perform very well. From this study, we can conclude that a bounded model checker seems to be a suitable tool for test case generation from formal specifications; it scales well to systems of industrial relevance, it generates the shortest possible test cases, and it is fully automated. Although there are some drawbacks – if the shortest test cases needed to cover a specific feature in the model is longer than the search depth of the bounded model checker, we have no way of telling if the test case simply do not exist or if it is longer than the search depth. That determination will have to be done through a manual process. However, based on the advantages bounded model checking offers, we will use it for the purpose of test-data generation in the domain of safety-critical reactive systems.

3.6 Chapter Conclusion

In this chapter, we discussed the use of formal specification languages to specify the behavior of reactive systems, in the context of specification-based testing. In this dissertation, we focus on such behavioral specifications and use them as the source
artifact to generate test cases through the model checking mechanism. We discussed
the three popular strategies for model checking – (1) explicit state model checking,
(2) symbolic model checking, and (3) bounded model checking. We identified the
pros and cons of each of these approaches and chose the bounded model checking
mechanism to drive our test generation since it is most efficient and results in shorter
test cases that satisfies out test obligations. Our empirical work will draw on this test
strategy and be used to generate test cases satisfying various structural test-coverage
criteria of interest.
Chapter 4

Initial Study on Test Effectiveness

To address our research objectives we performed an experimental study investigating the fault finding aspect of various structural test-coverage criteria discussed in earlier sections. In our initial study, we conducted an experiment to determine the effectiveness of test sets generated to various specification-based structural test-coverage criteria proposed in the research literature. We used the test framework discussed earlier in Chapter 1 for automating the generation of test cases from formal specifications modeled in the RSML-e formalism. We used the bounded model checking capability of the model checker NuSMV [30] to generate coverage adequate test cases. Most of the material presented in the following sections of this chapter have been adapted from our papers that were presented at the HASE 2004 [37] and COMPSAC 2005 conferences [50].

4.1 Experimental Setup

To evaluate the effectiveness of the generated tests generated using the test framework as shown in Figure 4.1, we conducted a small case study comparing the fault-finding capabilities of tests automatically generated with a model checker to tests generated randomly [62].

We were interested in answering the following question:

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

To answer this question, we devised an experiment evaluating the fault finding capability of three commonly suggested specification-based structural coverage criteria, state, transition, and decision coverage.

In the experiment we define these test criteria on a formal specification of the system modeled using the specification language RSML−e [121], but the criteria are applicable without modification to a broad class of languages. An RSML−e model consists of state variables and a next state relation for these state variables (this can be viewed as state machines with transitions between the states). The next state relation defines under which conditions the state variables change value (the state machines changes state), and are given in terms of Boolean expressions involving variables and arithmetic, relational, or boolean operators.

We use Γ to represent a test-suite and Σ for the formal model. In the following definitions, a test-case is to be understood as a sequence of values for the input variables in the model Σ and the expected outputs and state changes caused by these
inputs. The sequence of inputs will guide \( \Sigma \) from its initial state to the structural element, for example, a transition, the test-case was designed to cover. A test-suite is simply a set of such test cases. The test-coverage criteria we used are defined as follows:

**Variable Domain Coverage:** (Often referred to as state-coverage.) Requires that the test set \( \Gamma \) has test-cases that enable each control variable defined in the model \( \Sigma \) to take on all possible values in its domain at least once.

**Transition Coverage:** Analogous to the notion of branch coverage in code and requires that the test set \( \Gamma \) has test-cases that exercise every transition definition in \( \Sigma \) at least once.

**Decision Coverage:** Each decision occurring in \( \Sigma \) evaluates to true at some point in some test-case and evaluates to false at some point in some other test case.

### 4.1.1 Case Example: The Flight Guidance System

The experimental subject used in the case study was a model of the mode-logic in a production sized flight-guidance system (FGS), written in the RSML\(^{-}\) language [121, 123]. A Flight Guidance System (FGS) 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 generate pitch and roll guidance commands to minimize the difference between the measured and desired state. The FGS can be broken down to 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 aircrafts current and desired state and compute the pitch and roll guidance commands. In this case study we have used the mode logic. Figure 4.2 illustrates a graphical view of a FGS in the NIMBUS environment. The primary modes of interest in the FGS are the hori-
horizontal and vertical modes. The horizontal modes control the behavior of the aircraft about the longitudinal, or roll, axis, while the vertical modes control the behavior of the aircraft about the vertical, or pitch, axis. In addition, there are a number of auxiliary modes, such as half-bank mode, that control other aspects of the aircraft’s behavior. The FGS mode-logic model we have used in the experiment is production sized. The model consists of 2564 lines of code in RSML and consists of 142 state variables. When translated to SMV it consists of 2902 lines of code and required 849 BDD variables for encoding in NuSMV. The FGS is ideally suited for test case generation using model checkers since it is discrete—the mode logic consists entirely of enumerated and Boolean variables.

4.1.2 Fault Injection

To provide targets for our testing effort, we created a collection of faulty specifications. We first reviewed the revision history of the FGS model to understand what types of faults were removed during the original verification process. We then implemented a random fault seeder as shown in Figure 4.4 to inject representative faults to create a suite of faulty specifications. The faults we seeded fell into the following four categories:

**Variable Replacement**: A variable reference was replaced with a reference to another variable of the same type.

**Condition Insertion**: A condition that was previously considered a “don’t care” (*) in one of the tables was changed to T (the condition is required to be true).

**Condition Removal**: A condition that was previously required to be true (T) or false (F) in a table was changed to “don’t care” (*).
**Condition Negation:** A condition that was previously required to be true (T) in a table was changed to false (F), or vice versa.
MACRO When_LGA_Activated() :

TABLE
Select_LGA() : T;
PREV_STEP(.LGA) = Selected : F;
Is_This_Side_Active : *; /* Was T */
END TABLE
END MACRO

Figure 4.3: An example fault seeded into the FGS model.

Figure 4.4: Fault seeding.

We used our fault seeder to generate 100 faulty specifications (25 for each fault class). As an example, Figure 4.3 shows a missing condition fault contained in macro When_LGA_Activated, the fault was created by changing the table from requiring the Boolean variable Is_This_Side_Active that was originally true to a “don’t care.”

4.1.3 Test Set Generation and Execution

We then performed the experiment by conducting the following steps as shown in Figure 4.5, for several different structural coverage criteria:

1. We used the original specification to generate a test-suite to a coverage criterion of interest, for example, transition coverage, and measured the effort involved.
2. We ran the test-suite on the 100 faulty specifications and recorded the number of faults revealed as well as the time required run the test-suite.
3. We used the same effort (the sum of the time used to generate as well as run the structural test) to generate and run a randomly generated test-suite. We generated this suite using a statistical testing tool also implemented as part of Nimbus.
4. Given the results of the previous steps, we compared the relative fault finding capability of the randomly generated tests versus the structural tests.
Table 4.1: Number of trap properties generated for each coverage criterion

<table>
<thead>
<tr>
<th>State</th>
<th>Transition</th>
<th>Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>246</td>
<td>342</td>
<td>424</td>
</tr>
</tbody>
</table>

Table 4.2: Average length of the test cases in the test-suites

<table>
<thead>
<tr>
<th>State</th>
<th>Transition</th>
<th>Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.000</td>
<td>1.127</td>
<td>1.125</td>
</tr>
</tbody>
</table>

4.2 Experimental Results

As mentioned in Section 4.1, we generated tests to provide the coverage discussed in the previous section. Table 4.1 summarizes the number of trap properties generated for each coverage criterion from the FGS specification. We then generated test sequences using our tools infrastructure. Experiences regarding the performance of this test generation is presented in [63]. From our previous study, of interest for this paper is the average length of the test cases generated (Table 4.2). We were somewhat surprised by the very short test cases. A closer examination of the flight guidance model reveals that the state variables in the model are highly interconnected (they can move from one value to any other value in one step) making it possible to, in most cases, cover the constructs of interest (states, transitions, and decisions) in one step. In addition, the model checking technology we used as a test-case generation engine—the bounded model checker built into NuSMV—is guaranteed to find the shortest possible counterexample. Thus, we were likely to find the “simplest” test case that will exercise the construct of interest.

The tests we generated were then executed on our fault seeded models and the
Table 4.3: Number of faults revealed by each test-suite

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Var. Repl.</td>
<td>21</td>
<td>14</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Cond. Ins.</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Cond. Rem.</td>
<td>15</td>
<td>4</td>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>Cond. Neg.</td>
<td>25</td>
<td>12</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>Total</td>
<td>66</td>
<td>32</td>
<td>58</td>
<td>56</td>
</tr>
</tbody>
</table>

fault finding capability compared to that of tests generated randomly using the same effort as expended on each structural test set. The results are presented in Table 4.3.

The results here only reflect the relative fault finding capability—we made no effort to determine how many of the seeded faults actually led to semantically different specifications. To our disappointment, the structural tests performed consistently worse than our random tests and we began an investigation to determine why.

4.3 Key Observations

Our experiment indicated that the common specification test-adequacy criteria we evaluated are woefully inadequate and are not likely to reveal faults in our formal specification. These perform poorly when compared to random tests that were generated using the same time and effort we spent in generating the structural test-suites. We identified two reasons for this inadequacy:

First, a flight control system consists of, among other things, two flight guidance systems operating as a pair, one active and one spare, and there is a synchronization interface between them that can be used to force the spare FGS into any state. Thus,
there are generally two ways decisions in the FGS can be made true and false; either
the FGS is active and the decision logic is used normally, or the FGS is a spare and
the FGS-FGS interface is used to command which conditions are true and false. The
model checker relentlessly exploited this interface to achieve desired coverage without
actually exercising the true decision logic in the model. This problem can be solved
by introducing invariant properties that assert that the FGS in question be active.
However, this requires apriori knowledge of the FGS synchronization interface.

Second, and more importantly, great care has to be taken when formalizing the
coverage criteria for use with a model-checker; when the criteria are used as test-case
generators rather then test-case metrics issues taken for granted when instrumenting
models and code, for example, that the program flow actually reaches the construct of
interest, must be made explicit [62]. In our study, one could argue that our formalized
criteria captured the “letter” of the informal criteria but failed to capture the “spirit”
of the criteria. Both problems pointed to dangers in the published structural coverage
criteria used in the study; it should not be that easy to “cheat” the criteria using
the synchronization interface and the criteria must be formalized with great care to
capture the true spirit of the criteria.

To illustrate one of the problems with condition based coverage criteria, consider
a small code fragments (a), (b), and (c) in Figure 4.6. For simplicity, we use C like
syntax in our example. In example (a), if we want to use a model-checker to find test
cases that provide decision coverage of the decision on statement 1, we can generate
two trap properties

$$G(!((b \text{ or } c) = 1)) \quad G(((b \text{ or } c) = 1))$$

asserting that the decision cannot be true or false respectively. The model checker
will now find counterexamples to these trap properties giving us two test cases that
make the decision true and false. For example, it may generate the two test inputs
(b, \neg c, \neg d) and (\neg b, \neg c, \neg d) that would achieve decision coverage of the decision on statement 1. Note here that there is no requirement that the outcome of the decision on statement 1 affects the flow of the program on statement 3. In fact, one could even interpret definitions of condition coverage to not even require that the decision under consideration is evaluated.

The two test inputs (b, \neg c, \neg d) and (\neg b, \neg c, \neg d) achieve decision coverage, but there is no requirement that the outcome of the decision on statement 1 affects the flow of the program on statement 3; in the definitions of decision coverage and MC/DC we have seen in the literature there have been no consensus on how the various decision coverage criteria shall be interpreted. Consider a minor modification of the example above where we break out the decision on line 1 to a Boolean function, see Figure 4.6(b). Here we still achieve decision coverage of the decision on line 2 (and the outcome of the decision is still irrelevant).

With a minor modification rearranging the decision on line 4 we are not changing the behavior of the program, see Figure 4.6(c). In this case, if the language has lazy-evaluation, \texttt{A()} will not get called and the decision (now on line 2) will not get evaluated. Technically, however, with a naïve interpretation of decision coverage the two test inputs (b, \neg c, \neg d) and (\neg b, \neg c, \neg d) could be viewed as still achieving decision coverage of the decision on line 2. Naturally, these tests will not reveal any faults in the function since it will never get executed. In our initial study, we formalized the notion of decision coverage in an overly simplistic way; we simply stated that each decision must evaluate to true and false, but we did not make it explicit that the decision actually must be invoked during the program execution. This is exactly the problem we face with the condition based coverage criteria we have considered, for example, decision coverage and MC/DC coverage. If the decision of interest is masked out or never evaluated, the test is not particularly useful. To our knowledge, the
Figure 4.6: Code using a variable (a) and a function (b) and (c) to structure a larger condition.
condition based coverage criteria required in practice (for example, in certification to DO-178B [110]) and used in previous studies do not require the outcome of a decision to actually influence the program flow.

To solve this problem, the structural coverage criteria must be formalized in such a way that control flow is taken into account; the formalization must, at a minimum, assure that somehow the decisions are invoked. We believe these findings generalize to other systems and formalisms and, therefore, newer and better specification based coverage criteria must be developed.

4.4 Criteria with Execution-Guarantee

A solution to our problem is of course to ensure that a test case generated to cover a specific construct forces the construct to be invoked when the test case is being executed. We refer to this generation policy as the execution guarantee policy. The challenge is now to coerce the model checker to generate test cases that will adhere to this policy. To enforce the execution guarantee policy during test generation, we automatically introduce additional Boolean execution-guarantee variables into the model we use as a basis for the test-case generation as well as the trap properties. More specifically, in the target model we introduce an execution-guarantee variable for each function defined in the source model; a variable that is only true when the function is invoked. This variable is used to instruct the model checker to only consider test cases where a decision is actually evaluated. We also enhance each trap property generated for decisions encapsulated in functions with a clause that ensures that a counter example is created only when the function in which the decision is encapsulated has been invoked. With the enhanced target model and the enhanced trap properties, the model checker can enforce the execution guarantee policy and generate test-cases that adhere to the definition of decision coverage and MC/DC
coverage as defined in DO-178B.

If we again consider the simple example we introduced in Figure 4.6(c). We can introduce an execution-guarantee variable $A_{\text{invoked}}$ for function $A()$. This variable would be initialized to false with a next state relation that sets it to true only when $A$ is called and false otherwise. In NuSMV syntax we would define

\[
\begin{align*}
\text{init}(A_{\text{invoked}}) & := 0; \\
\text{next}(A_{\text{invoked}}) & := \\
& \begin{cases} \\
\text{case } d : 1; \\
1 : 0;
\end{cases} \\
& \text{esac;}
\end{align*}
\]

To generate test cases for achieving decision coverage on the decision $(b \text{ or } c)$ appearing at line 2 in the source model, we generate the following trap properties.

1. $G(!((b \text{ or } c) = 1) \land (A_{\text{invoked}} = 1))$

2. $G(!((b \text{ or } c) = 0) \land (A_{\text{invoked}} = 1))$

Both properties contain the clause $(A_{\text{invoked}} = 1)$ to ensure that a counter example will lead to a state in which the decision of interest has the right truth value and $A()$ has actually been invoked; this test case will exercise the decision as expected. Naturally, the generation of the execution-guarantee variables will have to take into account nested if statements, lazy evaluation, nested function calls, etc., all of which are done by the model augmentor using a set of inference and evaluation rules. These rules are formally discussed in the next section in the context of a transition-based system composed of entirely boolean variables, for convenience of discussion.
4.5 Formalization of the Execution-Guarantee Policy

In this section we will provide a more formal treatment on the machinery for encoding the notion of execution guarantee as a set of evaluation and inference rules. The model augmentor is responsible for implementing these sets of rules, thereby enforcing the execution-guarantee policy on test execution.

The notion of execution-guarantee can be explained formally in the context of a purely boolean model i.e. a model consisting of entirely boolean variables. Following are some definitions we will use in defining these sets of rules:

We assume that the system state $\sigma$ is uniquely determined by the value of $n$ variables, $\{x_1, x_2, \ldots, x_n\}$, where each $x_i$ takes its value from its domain $D_i$. Thus, the reachable state space of the system is a subset of $D = D_1 \times D_2 \times \ldots \times D_n$. The system may move from one state to another subject to the constraints imposed by its transition relation. We assume that there is a transition relation for each variable, $x_i$, specifying how the variable may change its value as the system moves from one state to another.

The complete transition for a variable $x_i$, denoted $\delta_i$, is the disjunction of all boolean expressions $e_k$ that guard transitions for $x_i$. Thus, $\delta_i = \bigvee_{k=1}^{n_i} e_k$, where $n_i$ is the number of transitions for the variable $x_i$.

The transition relation $\Delta$, is the conjunction of the complete transitions of all the variables $x_1, \ldots, x_n$. Thus, $\Delta = \bigwedge_{i=1}^{n} \delta_i$.

We finally define a basic transition system $\Sigma = (D, \Delta, \rho)$, where $D$ represents the state-space of the system, $\Delta$ represents the transition relation, and $\rho$ characterizes the initial system states.

We use “a” to denote an atom, and “e” to denote a predicate. An atom is defined as a boolean expression that contains no Boolean operators. A predicate is Boolean expression that is composed of atoms and zero or more Boolean Operators. We
assume that the Boolean operators are AND, OR, and NOT.

Given a particular state $\sigma$ of the model, we can think about usage of an “atom” in an expression “e” as a mathematical function $\Psi$ that maps the 3-tuple $(\sigma, a, e)$ to $\{0,1\}$. The actual mechanics of the function $\Psi$ are captured by the following rules:

- A Set of Inference rules that determine usage of an ”atom” a in an ”expression” e, given a state $\sigma$ of the model $\Sigma$.
- A Set of rules that determine evaluation order of expressions occurring in conditionals (if-then-else).

Table 4.4 lists the inference rules which are defined using standard operational se-
Table 4.4: Inference rules for determining usage of atom “a” in a boolean expression “e”

<table>
<thead>
<tr>
<th>Rule 1</th>
<th>$\sigma, a \vdash e_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\sigma, a \vdash e_1 \land e_2$</td>
</tr>
<tr>
<td>Rule 2</td>
<td>$\sigma \vdash e_1 = T, \sigma, a \vdash e_2$</td>
</tr>
<tr>
<td></td>
<td>$\sigma, a \vdash e_1 \land e_2$</td>
</tr>
<tr>
<td>Rule 3</td>
<td>$\sigma, a \vdash e_1, \sigma, a \vdash e_2$</td>
</tr>
<tr>
<td></td>
<td>$\sigma, a \vdash e_1 \land e_2$</td>
</tr>
<tr>
<td>Rule 4</td>
<td>$\sigma, a \vdash e_1 = F, \sigma, a \vdash e_2$</td>
</tr>
<tr>
<td></td>
<td>$\sigma, a \vdash e_1 \lor e_2$</td>
</tr>
<tr>
<td>Rule 5</td>
<td>$\sigma, a \vdash e_1$</td>
</tr>
<tr>
<td></td>
<td>$\sigma, a \vdash \neg e_1$</td>
</tr>
<tr>
<td>Rule 6</td>
<td>$\sigma, a \vdash a$</td>
</tr>
</tbody>
</table>

In addition to these inference rules, we need a set of rules that determine evaluation order of expressions in conditionals. To do this we begin with defining the operational semantics of boolean expressions. The definition consists of two parts: One which defines the syntax of terms, which in our case are boolean expressions and boolean
constants true, and false. The other part are the evaluation rules which capture the order in which terms get evaluated. The “→” symbol in the rules is to be read as “evaluates to”. For instance, in the ERule-IFTRUE rule, we are saying that “if true then $e_1$ else $e_2$” evaluates to $e_2$. Both these parts are defined in Table 4.5 and Table 4.6 respectively.

4.6 Enhanced Coverage Criteria with Execution Guarantee

Given the notion of execution guarantee discussed in the earlier section, we can now accurately formalize the condition based coverage criteria as defined in standards for safety-critical systems, for example, DO-178B [110]. In the remainder of this paper, a test-case is to be understood as a sequence of values for the input variables in the model and the expected outputs and state changes caused by these inputs. A test-
suite is simply a set of such test cases. For the discussion in this paper we will use the following definition of coverage criteria providing execution-guarantee:

**Single decision uses:** A test-suite is said to achieve single decision use coverage of a given state based specification, if each decision executes and evaluates to true at some point in some test case and executes and evaluates to false at some point in some other test case in the test-suite.

**MC/DC with decision uses:** MC/DC coverage with decision uses captures the code-based coverage criterion called modified condition/decision coverage (MC/DC). MC/DC was developed to meet the need for extensive testing of complex boolean expressions in safety-critical applications [26]. MC/DC requires us to show the following: Every condition (clause) within the decision has taken on all possible outcomes at least once, and Every condition has been shown to independently affect the outcome of the decision. A test-suite is said to achieve MC/DC coverage with decision uses of a specification, if every condition within a decision evaluates to true at some point in some test case and evaluates to false at some point in some other test case in the test-suite, in such a way as to independently affect the control flow at that point and the decision in which the condition occurs is executed.

**Masking MC/DC with decision uses:** According to DO-178B, a condition is shown to independently affect the outcome of a decision by varying just that condition while holding fixed all other possible conditions. Masking is an alternative approach to show the independent effect of a condition on the decision’s outcome by relaxing the restriction on holding all other possible conditions fixed [82]. Masking MC/DC is therefore a weaker criterion than MC/DC where some conditions may mask the effect of other conditions. The definition for Masking
MC/DC with decision uses is the same as MC/DC with decision uses except for the way in which we are showing the independent effect on a condition on a decision’s outcome.

Our next step was now to validate whether these enhanced coverage criteria are better at fault finding than their counterparts that do not enforce the guarantee policy. To this end, we reran our experiments using the same setup. The next section discusses our experimental design and the subsequent results obtained from running our experiment.

4.7 Fault Finding Experiment with Execution Guarantee

We conducted an empirical study to determine the feasibility of generating test cases with execution-guarantee and to compare the effectiveness of criteria with and without execution guarantee. To this end, we formulated two hypotheses.

4.7.1 Hypotheses

Enforcing the execution-guarantee policy during test generation requires the introduction of additional state variables into the NuSMV model. Since model checking techniques are sensitive to the state space explosion problem, the introduction of these additional variables might adversely affect the cost for the search for counter examples in terms of time and space. To confirm/refute this theory, we form the following hypothesis about feasibility of using test coverage criteria providing an execution guarantee.

Hypothesis 1:

*It is too costly (in terms of time and memory usage) to use a test criterion pro-*
viding execution guarantee to drive test generation using model checking tech-
niques.

Since our new formalization of the criteria guarantees that the proper part of the
specification is exercised upon test execution, we can speculate that the test-suites
generated using such criteria will be better at finding faults than using coverage
criteria that are purely condition based. We therefore formulate a second hypothesis.

**Hypothesis 2:**

*Test suites generated from condition-based coverage criteria with execution guar-
antee perform better in terms of fault finding than those generated without exe-
cution guarantee.*

### 4.7.2 Experimental Results and Analysis

We used the same experimental setup as discussed in Section 4.1 and generated test-
suites using eight different kinds of condition-based test criteria. The results from
rerunning our experiment are presented in Table 4.7. Three of these criteria provide
execution guarantee (the criteria postfixed with ‘Usage’ in the table). As a baseline
for comparisons, we generated a random test-suite using the same amount of time as
that we used in generating tests for the MC/DC usage criterion.

The results from our experiment are presented in Table 4.7. In the study, we
generated test-suites using eight different kinds of condition-based test criteria. Three
of these criteria provide execution guarantee (the criteria postfixed with ‘Usage’ in the table). As a baseline for comparisons, we generated a random test-suite using
the same amount of time as that we used in generating tests for the MC/DC usage
criterion.

Table 4.7 presents the total time (Time) and the maximum memory footprint
(Memory) for the NuSMV model checker to generate a test-suite according to each of
Table 4.7: Test set generation times (using NuSMV) and Fault detection effectiveness for various criteria

<table>
<thead>
<tr>
<th>Test Criteria</th>
<th>Test Generation Stats</th>
<th>Detected Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Suite Size</td>
<td>Time</td>
</tr>
<tr>
<td>Random</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Variable Domain</td>
<td>100</td>
<td>81.61s</td>
</tr>
<tr>
<td>Transition</td>
<td>313</td>
<td>193.67s</td>
</tr>
<tr>
<td>Decision</td>
<td>435</td>
<td>511s</td>
</tr>
<tr>
<td>Decision Usage</td>
<td>478</td>
<td>2615s</td>
</tr>
<tr>
<td>Masking MC/DC</td>
<td>552</td>
<td>8427s</td>
</tr>
<tr>
<td>Masking MC/DC Usage</td>
<td>361</td>
<td>8651s</td>
</tr>
<tr>
<td>MC/DC</td>
<td>537</td>
<td>8542s</td>
</tr>
<tr>
<td>MC/DC Usage</td>
<td>334</td>
<td>8796s</td>
</tr>
</tbody>
</table>

The results from Table 4.7 show that augmenting the decision coverage criterion with the notion of execution-guarantee may significantly increase the cost for generating test-suites while the increased cost still falls within a practical range. Surprisingly, the results show that augmenting the MC/DC criterion and the masking MC/DC criterion with the notion of execution guarantee only slightly increases the time and memory usage for NuSMV to generate the test-suite. These results show that it is quite feasible to use the criteria with execution guarantee for test generation. Therefore, the first hypothesis is refuted.

Table 4.7 also shows the size of the test-suite generated for each criterion. Note that since no effort was made to generate minimized test suites, redundant test cases may be present in each of the generated test-suites. For different criteria, the number of redundant test cases in the generated test suites may vary. This could be a reason why the test-suites generated for the MC/DC criterion with execution guarantee and the masking MC/DC criterion with execution guarantee are smaller in size than the test-suites generated for the corresponding versions not requiring invocation of the
conditions. Nevertheless, all these test-suites are coverage adequate according to the definition of these criteria.

The number of faults (the total and the breakdown per fault type) that are detected by test-suites generated for each of the criteria is also presented in Table 4.7. The data from the table shows that a test-suite generated from an a criterion providing execution guarantee performs consistently better than that generated from the version without this guarantee. These results support the second hypothesis. Especially, the test-suites generated for MC/DC and masking MC/DC with execution guarantee are able to detect all the 72 faults in the specification. This result indicates that these two criteria can be effective at fault detection. In another related study [61], it was found that the test suites generated for these two test criteria were the least sensitive to the effects of test set reduction in terms of fault finding; this indicates a high quality of the individual test cases in these test-suites. This particular topic will be discussed in detail in the next chapter of this dissertation.

One can also observe that the test-suites generated for MC/DC and masking MC/DC with execution guarantee are exactly equivalent in terms of fault detection. This observation is in agreement with the result in [28], where the masking form of the MC/DC coverage criterion was found to provide equivalent error detection capability in comparison to the strict form of MC/DC.

4.8 Threats to Validity

In our initial study there are three threats to external validity that prevent us from generalizing our observations. First, and most seriously, we are using only one instance of a formal specification in our experiment. The characteristics of the FGS model, for example, which is entirely modeled using Boolean and enumerated variables, most certainly affects our results and makes it impossible to generalize the
results to systems that, for example, contain numeric variables and constraints. Second, we are using seeded faults in our experiment. Although we took great care in selecting fault classes that represented actual faults we observed during the development of the FGS model, fault seeding always leads to a threat to external validity. Finally, we only considered a single fault per model. Using a single fault per specification makes it easier to control the experiment. Nevertheless, we cannot account for the more complex fault patterns that may occur in practice. The question as to whether the observed effectiveness of stronger coverage measures such as MC/DC, being merely a manifestation of a larger test size as opposed to the inherent nature of the coverage criterion, poses a threat to the internal validity of the results. This is because one could argue that any reported differences in the effectiveness of the test sets generated to various test-coverage criteria are artifacts of test set size; since a stronger coverage criteria often requires a lot more test cases to satisfy it, and therefore detects more faults in the model.

The initial results from our pilot studies are encouraging, but these results represent only a small step towards answering the question as to how effective these test criteria are in the specification-based testing domain. Clearly, one needs to address the threats to validity in our studies that are outlined above in order to generalize the applicability of our observations on the effectiveness of various structural test-coverage criteria in the specification-based testing domain.

Hence, there is need for careful and controlled experimentation, achieved through formal empirical studies, using a variety of formal specifications having different characteristics than the one used in the pilot studies. The results from such an empirical study would help us get a better understanding of the relationship between structural test-coverage criteria and fault detection effectiveness and also help us understand how the results are affected by carefully controlled changes i.e. to determine
cause-effect relationships and correlation between fault finding effectiveness and test parameters such as test case length, test set size etc. in the formal specification-based testing domain.

4.9 Chapter Conclusions

To summarize, in a previous investigation we evaluated how well model checking techniques scaled when used for test case generation from formal specifications. [63]. Although the approaches scaled well, our experience from this experiment cast some doubts as to the effectiveness of the test cases generated. To evaluate the test-suites generated to structural specification coverage criteria—state, transition, and decision coverage—we conducted a follow-up experiment where we compared the fault finding capability of these test-suites with randomly generated test-data. To provide a fair comparison, we assured that the effort spent on structural tests was comparable to the effort spend performing random tests. To our disappointment, we found that the structural tests uniformly performed worse than randomly generated tests. The poor performance of the structural tests is, in this case study, related to two issues (1) the structure of the flight guidance system (FGS) under test and (2) a mismatch between the specification coverage criteria and the semantics of the specification language.

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

Our experiences from this experiment raises some concern about the use of automated test case generation from formal specifications. Effective test case generation clearly requires an intimate knowledge of the structure and behavior of the system under test so that the tester can, for example, block input channels such as the Transfer Switch to get better quality tests. The coverage criteria used in specification testing and specification based testing must also be refined to better fit the semantics of the specification languages and the structure of the models captured in these languages.

In particular, there is a need to include some notion of data-flow information in the condition based coverage criteria. To this end, we have developed an approach to provide the formalization of condition-based test-coverage criteria with an execution guarantee that instructs the model checker to generate test cases that can better exercise the conditions in a formal specification. We have also performed an empirical study on a production sized flight guidance system model to evaluate (a) the feasibility of using criteria with an execution guarantee for test generation, and (b) the effectiveness of the generated test-suites in terms of fault finding. The results show that test-data generators such as model checkers are able to generate, within a reasonable cost, test cases that provide coverage with an execution guarantee; test cases that are more effective at fault finding than test cases generated using purely condition-based coverage criteria. In fact, the test cases generated to MC/DC and masking MC/DC with execution guarantees detect all the faults we seeded randomly in our specification; these criteria show clear promise as a practical means for generating effective test cases in the critical systems domain. Due to some limitations of our study, we are not able to broadly generalize our results. Further empirical studies
need to be performed on more diverse subjects with substantially more number of seeded faults or even real faults which can then be used to determine the robustness of the criteria with an execution guarantee.
Chapter 5

Expanded Study on Test Effectiveness

The results from our initial studies described in Chapter 4 have demonstrated (much to our dismay) that automated tests generated to satisfy various specification-based test-coverage criteria proposed in the testing literature are clearly not effective at detecting realistic faults. In the first study, test suites generated to satisfy three structural coverage criteria (state, transition, and decision coverage) appeared to perform worse than random test suites of similar cost. In the extended study, we enhanced our test-coverage criteria with the notion of evaluation and used a broader spectrum of test-coverage criteria, including strong coverage measures such as MC/DC, and saw a marginal improvement for state, transition, and decision coverage. However, these consistently performed worse than random testing—Although, MC/DC seemed to perform slightly better than random test suites of similar cost. There were several limitations that were identified in those studies that prevented us from generalizing the results over the domain of safety critical systems. Firstly, both studies (the initial study and the extended study) were conducted using a single example which was known to contain some unusual features that impacted test generation. Since we used only one case example, we do not know if the results from these studies generalize to other models having different characteristics than the Flight Guidance System (FGS) model used in these studies. Secondly, our studies do not indicate if test suites satisfying strong test-coverage criteria such as MC/DC are truly effective, or if they are effective because MC/DC test suites are generally quite larger.
Our initial studies have clearly identified opportunities for further work in this domain. Given the ever increasing power of automated testing tools and the strong incentives in transforming test adequacy measures as test data generators in this domain, we strongly believe that the issues identified in our earlier studies merit further study. To this end, we conducted an expanded study to address the threats to external validity identified earlier and to confirm/refute the hypothesis that directed test set generation using structural coverage criteria are more effective than random tests of equal size. The expanded study discussed in this chapter has benefitted greatly from having several case examples that are more representative of those used in industrial avionics applications, more rigorous experimental design, and addition of less sensitive test oracles, which are more commonly used in practice. The software test infrastructure for the expanded study was built by the Safety Critical Systems Group at the University of Minnesota, primarily to perform further research on understanding the influence of other test artifacts on test effectiveness. During the course of those investigations there was new data gathered, which we will use to generalize our findings from our earlier studies. In the following sections, we will briefly summarize the study and compare/contrast the results from this study [89] with our previous studies, and discuss the broader implications this has for specification-based testing in this domain.

5.1 Experimental Setup

The study measures the fault finding effectiveness of automatically generated test suites satisfying two structural coverage criteria, branch coverage and Modified Condition Decision Coverage, relative to randomly generated test suites of the same size. Tests were generated to satisfy the two coverage criteria using both counterexample-based test generation (based on the NuSMV model checker [30]) and a random gener-
ation approach [49, 107]. The effectiveness of the resulting test suites—as determined by mutation testing [79]—were then compared against purely randomly generated test suites of equal size. They effectiveness was measured using two expected value test oracles: an output-only test oracle that defines expected values for all outputs and a maximally powerful test oracle that defines expected values for all outputs and internal variables.

The study also measures the fault finding effectiveness of randomly generating test suites that are reduced with respect to the coverage criterion, relative to purely randomly generated test suites. This would give us valuable insight into the value of the coverage criterion: if tests randomly generated and reduced with respect to the coverage criterion are more effective than purely generated tests, we can conclude that the use of the coverage criterion led to the improved effectiveness.

In the following subsections we will formulate our research questions and describe our empirical evaluation in greater depth.

5.1.1 Experimental Hypothesis

Two structural coverage criteria are of interest in the expanded study: Branch coverage, and coverage [94, 26]. Branch coverage is commonly used in studies in testing research and improving branch coverage is a common goal in automatic test generation. Its effectiveness is therefore of interest to other testing researchers. coverage is a more rigorous coverage criterion based on exercising complex Boolean conditions (such as the ones present in avionics systems), and is required when testing critical avionics systems. Accordingly, we view it as likely to be an effective criterion, particularly for the class of systems studied in this report, and note the practical importance of the criterion [110].

In all, three independent variables were identified for the study: the structural
coverage criteria, the method of test generation, and the test oracle. We formulate our research questions as follows:

**RQ1:** Are test suites directly generated to satisfy branch and coverage more effective than randomly generated test suites of equal size?

**RQ2:** Are random test suites reduced to satisfy branch and coverage more effective than purely randomly generated test suites of equal size?

We attempt to investigate these questions using the experimental design discussed in the following sections.

### 5.1.2 Experimental Overview

In the study, we have used four industrial synchronous reactive systems developed at by Rockwell Collins engineers for business and regional (BRS) and air-transport aircraft, and a fifth synchronous reactive system developed as a case example at NASA. The Rockwell Collins systems were modeled using the Simulink notation and the NASA system using Stateflow (both tools from Mathworks Inc. [1, 2]), and were translated to the Lustre synchronous programming language [55] to take advantage of existing automation. Two of these systems, DWM 1 and DWM 2, represent portions of a Display Window Manager for a commercial cockpit display system. The other two systems—Vertmax.Batch and Latctl.Batch—represent the vertical and lateral mode logic for a Flight Guidance System (FGS). The NASA system, Docking.Approach, describes the approach behavior of the space shuttle to dock with the international space station.

Information related to these systems is provided in Table 5.1. We list the number of Simulink subsystems, which are analogous to functions, the number of blocks, which are analogous to operators, for the avionics systems developed in Simulink. We list
the number of states, transitions, and variables for the NASA example developed in Stateflow.

<table>
<thead>
<tr>
<th></th>
<th># Simulink Subsystems</th>
<th># Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>DWM_1</td>
<td>3,109</td>
<td>11,439</td>
</tr>
<tr>
<td>DWM_2</td>
<td>128</td>
<td>429</td>
</tr>
<tr>
<td>Vertmax_Batch</td>
<td>396</td>
<td>1,453</td>
</tr>
<tr>
<td>Latctl_Batch</td>
<td>120</td>
<td>718</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th># Stateflow States</th>
<th># Transitions</th>
<th># Vars</th>
</tr>
</thead>
<tbody>
<tr>
<td>Docking_Annouc</td>
<td>64</td>
<td>104</td>
<td>51</td>
</tr>
</tbody>
</table>

Table 5.1: Case Example Information

For each case example, we performed the following steps, as shown in Figure 5.1:

**Generated mutants:** We generated 250 mutants, each containing a single fault, and removed functionally equivalent mutants. (Section 5.1.3.)

**Generated structural tests:** We generated test suites satisfying branch and coverage using counterexample-based test generation. (Section 5.1.4.)

**Generated random test data:** We generated 1,000 random tests of test lengths between 2-10 steps. (Section 5.1.4.)

**Reduced test suites:** We generated reduced test suites satisfying branch and coverage using the test data generated in the previous two steps. (Section 5.1.5.)

**Random test suites:** For each test suite satisfying a coverage criterion, we created a single random test suite of equal size. In addition, we created test suites of sizes evenly distributed from sizes 1 to 1,000. (Section 5.1.5.)

**Computed fault finding:** We computed the fault finding effectiveness of each test suite using both an output-only oracle and an oracle considering all outputs and internal state variables (a *maximally powerful oracle*). (Section 5.1.6.)
5.1.3 Mutant Generation

We created 250 mutants (that is, faulty implementations) for each case example by introducing a single fault into the correct implementation. Each fault was seeded
into the program by either inserting a new operator into the system or by replacing an existing operator or variable with a different operator or variable. The mutation operators used in this study are fairly typical for mutation testing and are discussed at length in [4]. These mutation operators are similar to those used by Andrews et al. where they conclude that mutation testing is an adequate proxy for real faults [76].

We generated mutants such that the fault ratio—the number of mutants generated for a specific fault class versus the total number of mutants possible for that fault class—for each class of faults was approximately uniform. For example, assume for some example that there are $R$ possible Relational faults and $B$ possible Boolean faults. For a uniform fault ratio, we would seed $x$ relational faults and $y$ boolean faults in the implementation so that $x/R = y/B$.

One risk of mutation testing is semantically equivalent mutants—the scenario in which faults exist, but these faults cannot cause a failure (an externally visible deviation from correct behavior). This presents a problem when using oracles considering internal state: we may detect failures that can never propagate to the output. For our study, we used NuSMV to detect and remove functionally equivalent mutants for the four Rockwell Collins systems\textsuperscript{1}. This is made possible thanks to our use of synchronous reactive systems as case examples—each system is finite, and thus determining equivalence is decidable\textsuperscript{2}.

The Docking\textunderscore Approach system is also finite, but the number of potential non-equivalent mutants was much higher (i.e., mutants where testing led to no observable failure in an output). The cost of determining non-equivalence became prohibitive, and we therefore report only the results using the output-only oracle. Therefore, for every mutant reported as killed in our study, there exists at least one trace that

\textsuperscript{1} The percentage of mutants removed is very small, 2.8\% on average

\textsuperscript{2} Equivalence checking is fairly routine in the hardware side of the synchronous reactive system community; a good introduction can be found in [40].
can lead to a user-visible failure, and all fault finding measurements indeed measure actual faults detected.

5.1.4 Test Data Generation

We generated a single set of 1,000 random tests for each case example. Each individual test in these sets contains between 2 and 10 steps with the number of tests of each test length distributed evenly in each set of tests. We refer to this as a random test suite.

We have also generated test suites satisfying the branch and [94, 82, 28]. Several variations of exist—for this study, we use Masking, as it is a common method of computing within the avionics community [28].

We use a counterexample-based test generation approach to generate tests satisfying these coverage criteria [49, 107]. This approach is guaranteed to generate a test suite achieving the maximum possible coverage. We have used the NuSMV model checker in our experiments [30] because we found it efficient and have found the tests produced to be both simple and short [62].

As all of our case examples are modules of larger systems, the tests generated are effectively unit tests.

5.1.5 Test Suite Reduction

Counterexample-based test generation results in a separate test for each coverage obligation. This results in a large amount of redundancy in the tests generated, as each test likely covers several coverage obligations. Consequently, the test suite generated for each coverage criterion is generally much larger than is required to provide coverage. Given the correlation between test suite size and fault finding effectiveness [95], this has the potential to yield misleading results—an unnecessarily large test suite
may lead us to conclude that a coverage criterion has led us to select effective tests, when in reality it is the size of the test suite that is responsible for its effectiveness. To avoid this, we reduce each naively generated test suite while maintaining the coverage achieved. To prevent us from selecting a test suite that happens to be exceptionally good or exceptionally poor relative to the possible reduced test suites, we produce 50 different test suites for each case example using this process.

Per RQ2, we also create tests suites satisfying branch and coverage by reducing the random test suite with respect to the coverage criteria. Again, we produce 50 tests suites satisfying each coverage criterion.

For both random and counterexample-based test generation, reduction is done using a simple greedy algorithm. We begin by determining the coverage obligations satisfied by each test generated, and initialize an empty test set reduced. We then randomly select a test input from the full set of tests; if it satisfies obligations not satisfied by any test input in reduced, we add it to reduced. We continue until all tests have been remove from the full set of tests.

For each test suite reduced to satisfy a coverage criteria, we produce a reduced test suite of equal size using the set of random test data. We measure test suite size in terms of the number of total test steps, rather than the number of tests, as random tests are on average longer than tests generated using counterexample-based test generation. These random test suites are used as a baseline when evaluating the effectiveness of test suites reduced with respect to coverage criteria.

We also generate random test suites of sizes varying from 1 to 1,000. These tests are not part of our analysis, but help provide context in our illustrations.
5.1.6 Computing Fault Finding

All test suites are executed on their corresponding mutant sets and original case example. We recorded the value of every internal and output variable at each step of every test.

In our study, we use expected value oracles, which define concrete expected values for each test input [85]. We explore the use of two oracles: a output-only oracle that defines expected values for all outputs, and a maximum oracle that defines expected values for all outputs and all internal state variables.

To determine the fault finding of a given test suite $t$ and an oracle $o$ on a case example, we simply compare the values produced by the original case example against each mutant using (1) the subset of the full test suite corresponding to the test suite $t$ and (2) the subset of variables for oracle $o$. The fault finding effectiveness of the test suite and oracle pair is computed as the number of mutants detected (or “killed”). We use the information produced by this analysis to evaluate our research questions.

5.2 Experimental Results and Key Observations

The fault finding results from this study are shown in Tables 5.2 and 5.3, listing for each case example, coverage criterion, test generation method, and oracle: the average fault finding for test suites reduced to satisfy a coverage criteria, next to the average fault finding for random test suites of equal size; the relative change in average fault finding when using the test suite satisfying the coverage criteria versus the random test suite of equal size. Note that negative values for \% Change indicate the test suites satisfying the coverage criteria are less effective on average than random test suites of equal size. Since the data listed in Tables 5.2 and 5.3 do not fit any known probability distribution, a non-parametric test was used for statistically evaluating
Table 5.2: Average fault-finding effectiveness results, branch coverage criterion. OO = Output-Only, MX = Maximum

The statistical analysis of the results was performed using a confidence interval of 95%, and the results show that for both coverage criteria, the automatically generated test suites perform significantly worse than random test suites of equal size when coupled with an output-only oracle (see Figure 5.2). On the other hand in case of the Docking Approach case example (see Figure 5.3), test suites generated to satisfy structural coverage perform dramatically better, finding approximately 16 times as

Table 5.3: Average fault-finding effectiveness results, MCDC criterion. OO = Output-Only, MX = Maximum

The statistical analysis of the results was performed using a confidence interval of 95%, and the results show that for both coverage criteria, the automatically generated test suites perform significantly worse than random test suites of equal size when coupled with an output-only oracle (see Figure 5.2). On the other hand in case of the Docking Approach case example (see Figure 5.3), test suites generated to satisfy structural coverage perform dramatically better, finding approximately 16 times as
many faults as random test suites of equal size. When using the maximum oracle, we see that test suites generated via counterexample-based test generation fare better. In some instances, these tests outperform random test suites of equal size (most notably, the Vertmax_Batch case example). The effectiveness of these test suites when using the maximum oracle indicates that they are indeed effective at uncovering faults, but are not effective at causing those faults to propagate to the outputs—erroneous internal values are being produced, but getting *masked* out. This is a result of generating tests which exercise syntactic elements within the source code. Such tests manipulate internal state variables as required to satisfy the criterion, but do not propagate the resulting values to the outputs. By pairing the test suite with a stronger oracle monitoring internal state, masking becomes irrelevant (as we are monitoring all internal state values) and our fault finding effectiveness improves accordingly.

Furthermore, randomly generated test suites reduced while maintaining structural coverage find more faults than pure randomly generating test suites of equal size. This is turn leads us to conclude that the improved effectiveness of the coverage adequate test sets was due to the use of the coverage criterion.

<table>
<thead>
<tr>
<th></th>
<th>Branch Coverage</th>
<th>MCDC Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>DWM_1</td>
<td>100.0%</td>
<td>100.0%</td>
</tr>
<tr>
<td>DWM_2</td>
<td>100.0%</td>
<td>97.76%</td>
</tr>
<tr>
<td>Vertmax_Batch</td>
<td>100.0%</td>
<td>99.4%</td>
</tr>
<tr>
<td>Latctl_Batch</td>
<td>100.0%</td>
<td>100.0%</td>
</tr>
<tr>
<td>Docking_Approach</td>
<td>58.1%</td>
<td>37.76%</td>
</tr>
</tbody>
</table>

Table 5.4: Coverage Achieved (of Maximum Coverage) by Randomly Generated Test Suites Reduced to Satisfy Coverage Criteria
Figure 5.2: Fault finding effectiveness using NuSMV-generated test suites (‘+’), randomly generated test suites reduced to satisfy a coverage criterion (‘o’), and pure random test generation (line). Output-only oracles.

5.3 Discussion of Results and Implications for Model-Based Testing

We can draw two key conclusions from the reported results in this study. First, automated test generation to satisfy branch or MC/DC coverage does not, for the systems
Figure 5.3: Fault finding effectiveness using NuSMV-generated test suites (‘+’), randomly generated test suites reduced to satisfy a coverage criterion (‘o’), and pure random test generation (line). Maximum oracles.

investigated, yield effective tests relative to their size. This indicates that satisfying even a highly complex coverage criterion such as MC/DC is a poor indication of test suite quality. Second, the use of coverage measures in conjunction with random testing results in more effective test suites than pure random testing alone. These results present a serious problem in the context of automatic test generation and are quite
troublesome given the mandated use of test-adequacy measures in avionics systems and the tendency to use them to auto-generate test cases. These results indicate that we cannot blindly trust test automation and the current methods of determining test suite adequacy in such systems are by themselves inadequate.

As all our empirical studies on test effectiveness indicate, there are a variety of factors, such as the formulation of structural coverage criteria, structure of the case examples, and the behavior of our test generation mechanism (NuSMV) itself that lead to the lack of effectiveness of the test suites generated via counterexample-based methods. We have shown previously in our empirical studies how the structure of the program can lead our test generation mechanism to produce ineffective tests due to the masking present in such systems, where some expressions are prevented from influencing the outputs. This issue is further compounded by the fact that test inputs generated using counterexample-based generation tend to be short, and manipulate only a handle of values. Although, such tests satisfy the coverage obligations, they do not consider the propagation of errors to the output i.e., they are not effective at causing those errors to propagate to an observable output—erroneous internal states are being produced, but getting masked out. In contrast, random tests vary in length and have a better chance (as demonstrated in both our studies and the expanded study) to overcome the masking problem and provide better fault detection capability.

As highlighted by the Docking Approach case example, however, tests generated to satisfy structural coverage can sometimes dramatically outperform random test cases. This example differs from the Rockwell Collins systems primarily in the way it is structured: large portions of the system’s behavior are activated only when very specific conditions are met. The state space is both deep and contains bottlenecks; this is because the system behavior requires a particular sequence of actions to occur before the space shuttle can dock with the international space station. If, at any
point in the sequence, the condition is not satisfied then it reverts to the initial state. This requires relatively long tests with specific combination of inputs—something that the structural tests are adept at. Random testing on the other hand, is highly unlikely to reach much of the state space in this case. By contrast, the other case examples from Rockwell Collins do not have such a structure—the state space in those systems is shallow and highly interconnected and is, therefore, easier to cover with random tests.

The results have provided us with solid statistical evidence confirming our earlier findings and refuting our original hypothesis that automated test generation using structural coverage criteria can lead to effective tests. Such a finding has several implications for practitioners in this domain. First, transforming test adequacy measures for auto generating test suites may not be the right thing to do without accounting for other test factors such as the choice of the test oracle, the program structure, and finally the test generation mechanism. Second, the coverage criteria that were studied can be used in a supplemental manner to guide generation of random tests that are coverage adequate with respect to the coverage criterion. We have seen that random tests which are reduced with respect to MC/DC performed very well at fault detection when compared to purely generated random tests.

These findings have broader implications when determining the adequacy of the testing effort itself. The coverage criteria are by themselves weak and lead to ineffective tests. One solution could be to improve our automated test generation tools to generate longer test cases, increasing the chances that a corrupted internal state propagates to the output. We could also augment our test-coverage criteria to take advantage of improved search heuristics available in current auto-test generation tools. Recent work has shown that it is possible to fine tune SAT solvers by exploiting the structure of the problem in order to increase efficiency [115]. Other notable
contributions include the pruning of the search space by reusing information across various SAT instances etc. [116, 115]. Since the auto-generated tests seem to be quite sensitive to the structure of the model, We could also leverage techniques and tools to visualize large state spaces and gain a better insight into the internal structure of the state space and choose the appropriate testing strategy [78]. In addition to this, we could also utilize current research on typical properties of state space parameters such as the diameter, number of strongly connected components, and the local structure of state spaces (clustering etc.) [101, 100]. Since model checkers treat state spaces as arbitrary graphs, these typical properties can be potentially exploited during model checking. For instance, the bounded model checking algorithms exploit the fact that state space diameter is typically small for small state spaces. Although it is expensive to compute the diameter and girth of an arbitrary graph, we can compute good estimates of these parameters with the use of depth-first or breadth-first search. Current model checkers can be improved to leverage such information in determining the test inputs and generating effective tests that are less prone to the way the program is structured.

The expanded study discussed in this chapter has served to further strengthen our central hypothesis that test adequacy measures are indeed weak when used as a basis for auto-test generation. Given the incentives auto-test generation offers in this domain, one needs to view the testing process holistically, considering the other factors that influence the quality of auto generated test sets. The expanded study in conjunction with our earlier work on test effectiveness has led researchers to further examine the influence of these test artifacts and has led to some significant increase in our understanding of test effectiveness in this domain.

Recent work in this area conducted by Staats et.al. [117] aims to explores the influence of multiple artifacts on the effectiveness of software testing. It highlights
both in theory and practice, how testing artifacts can interact in significant ways to influencing test effectiveness. It shows how program structure can strongly impact the influence of test oracles and the effectiveness of test inputs can be improved through the use of strong oracles. We have strong empirical evidence to back these findings. Program structure also influences the cost and effectiveness of structural coverage criteria. These findings provide a strong motivation to consider how such artifacts may influence results in an empirical study. Their work seems to indicate that we can potentially improve testing effectiveness when using structural coverage criteria by inlining the program structure and using the maximum oracle since it outperforms the output only oracle when using structural coverage criteria. However, as witnessed in the expanded study, the impact of the test oracle is significantly diminished when inlining the program structure, the structural test inputs improve, but the maximum oracle becomes weaker. These results from this recent work, thus highlight the importance of understanding the interaction between testing artifacts, quantify how these interactions occur in avionics software, and point towards future directions in testing research by providing an improved conceptual framework for discussing software testing and demonstrate that program structure strongly impacts both the cost and effectiveness of structural coverage criteria, with negative implications for current testing practice.

5.4 Chapter Conclusion

Our empirical work on Test Effectiveness highlights the pitfalls encountered in transforming test adequacy measures into test suite generators and highlight the need for more research into how the coverage criterion, test generation approach, and the structure of the system under test jointly influence the effectiveness of testing. Test practitioners in this domain need to be cognizant of this when using auto-generated
test suites to satisfy the coverage needed for certification purposes, particularly the avionics domain. In addition, our empirical work has laid the foundation for driving research into the influence of other test artifacts on test effectiveness in domain of safety critical systems.

The expanded study discussed in this chapter has validated the results from our earlier work by demonstrating the inadequacy of test-coverage criteria as test data generators in this domain. The findings from the study indicate a need for methods to determine test adequacy that not only provide the desired coverage, but also lend themselves as targets for automated test generation techniques. These criteria must address the problem holistically to account for all factors influencing the quality of testing, including the program structure, the nature of the state space of the System Under Test, the test oracle used, and finally, the test generation mechanism itself.
Chapter 6

Impact of Test Reduction on Test Effectiveness

As discussed in the previous chapters, Model checking techniques can be successfully employed as a test-case generation technique to generate tests from formal models. The number of tests-cases produced, however, is typically large for complex coverage criteria such as MC/DC. Test-suite reduction can provide us with a smaller set of test-cases that preserve the original coverage—often a dramatically smaller set. Nevertheless, one potential drawback with test-suite reduction is that this might affect the quality of the test-suite in terms of fault finding. The effect of test-suite reduction on fault finding capability has not been extensively studied in test literature and previous empirical studies provide conflicting evidence on this issue. However, such experimental data, if made available, can be used to determine the robustness of the various specification-based structural test-coverage criteria in terms of their sensitivity at fault detection, to test conditions such as test-suite reduction.

In the following sections of this chapter, we will describe our experiment, analyze the results, and discuss the implications for testing based on formal specifications.

6.1 Hypotheses

To investigate the effects of test-suite reduction on the fault finding ability of auto generated tests from formal models of software, we performed an experiment using a large case example of a Flight Guidance System, generated reduced test-suites for a
variety of structural coverage criteria while preserving coverage, and recorded their fault finding effectiveness.

We designed our experiment to test the following two hypotheses:

Hypothesis 1: Test reduction of a naively generated specification based test-set can produce significant savings in terms of test-set size.

Hypothesis 2: Test reduction will adversely affect the fault finding capability of the resulting test set.

We formulated our hypotheses based on two informal observations. First, in a previous study we got an indication that one could achieve equivalent transition and state coverage with approximately 10% of the full test-set generated [63], we believe this generalizes to other criteria as well. Second, intuitively, more tests-cases ought to reveal more faults. Only an extraordinarily good test adequacy criterion would provide a fault finding capability that is immune to variations in test-suite size, and we speculate that none of the known coverage criteria posses this property.

6.2 Experimental Setup

In our experiment the aim was to determine how well a test-suite generated to provide a certain structural or condition based coverage of a specification model reveals faults in the implementation of that model as compared to a reduced test-suite providing the same coverage of the original model. To provide realistic results, we conducted the experiment using the close to production model of the Flight Guidance System (FGS) discussed in Chapter 4.

We conducted the experiment through the five the steps as shown in Figure 6.2 and outlined below.
1. We used the original FGS specification to generate test-suites to various coverage criteria of interest, for example, transition coverage or MC/DC. Note here that we did this naïvely in that we generated a test-case for each construct.
we needed to cover. Thus, the test-suites were straightforward to generate and easily traceable back to the constructs of the model they were intended to cover, but the tests in each test-suite were also highly redundant in terms of the coverage they provided.

2. To provide targets for our conformance testing experiment, we used fault seeded models to emulate the actual implementation. We generated 100 faulty models of the FGS by randomly seeding one fault per faulty specification. The fault classes we seeded were derived based on the change history of the model and faults likely to be inserted when developing an implementation based on the specification; these have been previously discussed in our initial study on test effectiveness. By manually inspecting the resultant mutant models, we found that 28 of them were semantically identical to the original FGS model. Therefore, our study uses the remaining 72 mutants that contain real faults as targets of the testing effort.

3. We ran the full (non-reduced) test-suite on the 72 faulty implementations and recorded the number of faults revealed.

4. We generated and ran five reduced test-suites for each full test-suite, ensuring that the desired coverage criterion was maintained. As discussed below, since the test reduction algorithm picks test-cases at random from the complete test-suite we generated several reduced sets for each full test-suite to avoid skewing our results because we were lucky (or unlucky) in the selection of tests for a reduced test-suite. We arbitrarily chose to generate five reduced test-suites to provide higher confidence in our results while at the same time keep the effort conducting the experiments manageable.

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

The details of the steps are described in the following sections.

6.2.1 Fault Injection and Detection

To provide targets for our testing effort we needed a collection of fault implementations. Since the structure of an RSML\textsuperscript{−e} model is basically a collection of nicely formatted nested if-statements where each condition guards the assignment of a state variable, an implementation of the model will have a very similar structure to the model from which it was developed. Consequently, the fault classes applicable in the implementation domain are similar to the ones in the model domain. In addition, we have extensive support to execute, manipulate, and measure test-coverage over RSML\textsuperscript{−e} models. Therefore, we decided to use a collection of fault seeded RSML\textsuperscript{−e} models to emulate the implementations rather than implementing the models in, for example, C, and seed faults in the resultant implementations (we would simply seed faults from the same fault classes in an implementation in C rather than an “implementation” in RSML\textsuperscript{−e} ).

To create the faulty specifications, we first reviewed the revision history of the FGS model to understand what types of faults were commonly made when developing models. We also studied how the models would be implemented in source code to determine which faults seemed likely to be inserted in a manual implementation. Note here that our focus has been on faults inserted during manual coding, not the types of faults that might be inserted by a code-generator. Nevertheless, given the limited types of faults possible when deriving an implementing from a model expressed in a language such as RSML\textsuperscript{−e}, we believe that the faults injected by a code-generator would come from the same fault classes—the main difference from manual coding would be that a code-generator would presumably inject the faults systematically
rather than spuriously as injected by a programmer.

The faults that we identified as commonly inserted during the FGS development effort as well as likely faults to be injected during implementation were the same as the fault classes that were identified and described in our initial study on test effectiveness.

Since we focused on likely faults, we have omitted some possible fault classes, for example, missing state variables and misdirected or missing transitions. In our experience, such severe faults were uncommon and so obvious they would be quickly eliminated and, thus, would not linger until testing commenced.

We used our fault seeder to generate 100 faulty models to use as test targets (25 for each fault class). By manually inspecting the resulted faulty models, we found that 28 of them were semantically identical to the original FGS model. Therefore, our study uses the remaining 72 mutants that contain faults detectable through testing.

During this testing experiment, we used a quite sensitive oracle to determine if a test-case revealed a fault. Given the input sequence of a test-case, we compared both the generated output as well as the internal state of the model to determine if a fault was present. In general, the internal state information of the system under test may not be available to the oracle. Thus, our oracle was able to detect faults that may not have manifested themselves as erroneous outputs, but only as a corrupt system state. We chose this approach since we expect this to be the type of oracle used when performing conformance testing of auto-generated code; we have long advocated that the auto-generated code should conform to coding standards that make this type of traceability between specification and implementation feasible [128].
6.2.2 Test Set Generation and Reduction

We generated full test-suites using the model checker as the test-data generator – an approach discussed in Chapter 3. We used the NIMBUS to translate to the input language of the model-checker NuSMV [30] and also to generate the trap properties corresponding to the test-coverage criteria discussed above. The model and the trap properties are then given to the NuSMV tool to create the full test-suites.

A single test-case usually satisfies more than one test obligation. For instance, a test-case used to cover a certain state of interest may also cover other states during its execution. This then provides for a way to reduce the size of the final test-suite by choosing a subset of test-cases that preserves the coverage obtained by the full test-suite.

Finding a minimal test-suite that satisfies the test requirements is in general an NP problem [48], but often greedy heuristics suffice to generate significantly reduced test-suites. The method we use begins with an empty set of test cases and initializes the coverage to zero (Figure 6.2). The greedy algorithm then randomly picks a test-case from the full test-suite, runs the test, and determines if the test-case improved the overall coverage (for whatever criterion in which we are interested). Any test-case that improves the coverage is added to the reduced set. This continues until we have exhausted all the test-cases in the full test-suite—we now have a, hopefully, much smaller suite that has the same coverage as the full test-suite.

The test case minimization approach we employ is a post processing technique to eliminate redundant test-cases. This means that we first generate the entire test-suite that satisfies a particular test criterion and then perform minimizations on it. If the goal was to get small test-suites it would be better to perform this filtering as the test-cases are generated. Nevertheless, since we also need the full test-suite for our experiment we choose this post processing approach.
Algorithm 6.2.1: TEST-REDUCE(Σ, Γ, η)

INPUTS:
Model Σ, test suite Γ, and test criterion η

OUTPUT:
Reduced test set Ω

Ω ← ∅; ReducedTest set
AC ← 0; Actual Coverage
PC ← 0; Previous Coverage
shuffle(Γ);
repeat
choose a test case f from Γ;
run f against the model Σ;
Measure actual coverage AC;
if AC ≠ PC
then Ω ← Ω ∪ \{f\};
PC ← AC;
until Γ is exhausted
return (Ω);

Figure 6.2: Algorithm for test-suite reduction.
<table>
<thead>
<tr>
<th>Test Criteria</th>
<th>Size</th>
<th>VR</th>
<th>CN</th>
<th>CI</th>
<th>CR</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>100</td>
<td>21</td>
<td>25</td>
<td>5</td>
<td>15</td>
<td>66 (92%)</td>
</tr>
<tr>
<td>Variable Domain</td>
<td>115</td>
<td>14</td>
<td>15</td>
<td>2</td>
<td>4</td>
<td>32 (44%)</td>
</tr>
<tr>
<td>Transition</td>
<td>313</td>
<td>20</td>
<td>24</td>
<td>5</td>
<td>15</td>
<td>64 (89%)</td>
</tr>
<tr>
<td>Decision</td>
<td>435</td>
<td>23</td>
<td>24</td>
<td>5</td>
<td>15</td>
<td>67 (93%)</td>
</tr>
<tr>
<td>Decision Usage</td>
<td>478</td>
<td>23</td>
<td>24</td>
<td>7</td>
<td>15</td>
<td>69 (96%)</td>
</tr>
<tr>
<td>MC/DC</td>
<td>537</td>
<td>22</td>
<td>25</td>
<td>7</td>
<td>16</td>
<td>70 (97%)</td>
</tr>
<tr>
<td>MC/DC Usage</td>
<td>334</td>
<td>23</td>
<td>25</td>
<td>8</td>
<td>16</td>
<td>72 (100%)</td>
</tr>
</tbody>
</table>

Table 6.1: Full test set generation for various criteria along with their fault detection capability

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

6.3 Key Observations

As a baseline for our experiments, we ran the full test-suites to determine their fault finding capability. To get a basic idea of their fault finding capability, we also created a collection of randomly generated tests to us as a comparison. We expended the same amount of time automatically generating and running the random tests as we did running the tests providing transition coverage. Thus, the randomly generated tests serve as a simple baseline for the other test-suites; one would expect the tests carefully crafted to provide a certain coverage to perform better than the randomly generated test-set. The results are summarized in Table 6.1. The table shows the
number of test-cases in each test-suite and their fault finding capability (total fault finding capability as well as broken down per fault-class).

As can be seen, the randomly generated test perform surprisingly well compared to the test-suites providing structural coverage. Note here that the randomly generated tests were of the length 100 input steps whereas the tests generated to a specific coverage were typically very short (1-3 steps). Nevertheless, the comparison is still relevant since we were not interested in comparing the number of test-cases but rather the effort involved in generating the tests; given equivalent effort in test-generation and test-execution, random testing performed very well. We have discussed the reasons behind the poor performance of Variable Domain and Transition Coverage in a previous study [62] and a discussion of this topic is outside the scope of this paper.

From the results in Table 6.1 one can also observe that the more rigorous the test criteria, the better the fault finding capability. For instance, for this particular case-example MC/DC with usage detects all faults yielding a behaviorally different model.

6.3.1 Test-Suite Reduction

As mentioned earlier, we generated five different reduced test-suites for each coverage criterion to control the possibility that we by chance got a very “good” or very “poor” reduced test-suite. The results of the reduction algorithm can be seen in Table 6.2.

The results support our first hypothesis that test reduction results in significant savings in terms of test-suite size. In all cases there was at least an 80% average reduction in the size of the test-suite. This reduction reinforces the findings in [129, 130, 109, 81] and is to be expected since our test-case generation method produces one test-case per construct of interest (variable value, transition, decision value, or MC/DC value). This generation approach is desirable since it makes traceability of
<table>
<thead>
<tr>
<th>Criteria</th>
<th>Full</th>
<th>r.1</th>
<th>r.2</th>
<th>r.3</th>
<th>r.4</th>
<th>r.5</th>
<th>Avg.</th>
<th>Red.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Var. Domain</td>
<td>115</td>
<td>19</td>
<td>22</td>
<td>18</td>
<td>21</td>
<td>21</td>
<td>20.2</td>
<td>82%</td>
</tr>
<tr>
<td>Transition</td>
<td>313</td>
<td>35</td>
<td>43</td>
<td>29</td>
<td>38</td>
<td>43</td>
<td>37.6</td>
<td>88%</td>
</tr>
<tr>
<td>Decision</td>
<td>435</td>
<td>45</td>
<td>44</td>
<td>44</td>
<td>45</td>
<td>42</td>
<td>44.0</td>
<td>90%</td>
</tr>
<tr>
<td>Decision Usage</td>
<td>478</td>
<td>37</td>
<td>43</td>
<td>47</td>
<td>43</td>
<td>38</td>
<td>41.6</td>
<td>91%</td>
</tr>
<tr>
<td>MC/DC</td>
<td>537</td>
<td>34</td>
<td>33</td>
<td>29</td>
<td>34</td>
<td>32</td>
<td>32.4</td>
<td>94%</td>
</tr>
<tr>
<td>MC/DC Usage</td>
<td>334</td>
<td>30</td>
<td>30</td>
<td>33</td>
<td>32</td>
<td>33</td>
<td>31.6</td>
<td>91%</td>
</tr>
</tbody>
</table>

Table 6.2: Reduced test set sizes for various test reduction runs. r.1–r.5 indicate the size of the five reduced test-suites. Avg. denotes the average reduction of the five.

the test-cases to their test-objective straightforward (one-to-one mapping). On the other hand, as is evident from Table 6.2, it leads to a significant number of overlapping (with respect to coverage) test-cases that may or may not add to the fault-finding ability of the test-suite; this is the topic of the next section.

6.3.2 Effect on Fault Detection Effectiveness

The fault finding capability of the full as well as reduced test-suites is summarized in Table 6.3. The results are in agreement with our second hypothesis that test-suite reduction will adversely impact the fault finding ability of test-suites that are derived from synchronous data-flow models.

As shown in Table 6.3, the number of faults detected by the reduced test-suites is significantly less for all coverage criteria that were examined in our experiment; in all cases there was at least a 7% reduction in the fault detection effectiveness. One may argue that a 7% reduction is rather small, but for our domain of interest, automated code generation in critical systems, any reduction in fault finding ability is unacceptable.
Table 6.3: Fault finding capability of the reduced test-sets. r.1–r.5 indicate the
number of faults found with each of the five reduced test-suites. Avg. denotes the
average number of faults found with the five reduced test-suites.

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Full Set</th>
<th>r.1</th>
<th>r.2</th>
<th>r.3</th>
<th>r.4</th>
<th>r.5</th>
<th>Avg.</th>
<th>Red.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Var. Domain</td>
<td>32</td>
<td>28</td>
<td>29</td>
<td>25</td>
<td>28</td>
<td>25</td>
<td>27.0</td>
<td>15.6%</td>
</tr>
<tr>
<td>Transition</td>
<td>64</td>
<td>58</td>
<td>58</td>
<td>58</td>
<td>59</td>
<td>57</td>
<td>58.0</td>
<td>9.38%</td>
</tr>
<tr>
<td>Decision</td>
<td>67</td>
<td>62</td>
<td>61</td>
<td>62</td>
<td>62</td>
<td>61</td>
<td>61.6</td>
<td>8.06%</td>
</tr>
<tr>
<td>Dec. Usage</td>
<td>69</td>
<td>62</td>
<td>63</td>
<td>63</td>
<td>62</td>
<td>63</td>
<td>62.6</td>
<td>9.28%</td>
</tr>
<tr>
<td>MC/DC</td>
<td>70</td>
<td>64</td>
<td>63</td>
<td>63</td>
<td>63</td>
<td>63</td>
<td>63.2</td>
<td>9.71%</td>
</tr>
<tr>
<td>MC/DC Usage</td>
<td>72</td>
<td>67</td>
<td>66</td>
<td>67</td>
<td>67</td>
<td>67</td>
<td>66.8</td>
<td>7.22%</td>
</tr>
</tbody>
</table>

From our results we can also observe that the most rigorous coverage criteria,
MC/DC with Usage, seems to be the least sensitive to the effect of test-suite reduc-
tion. We speculate that this is because it is simply harder to come up with a test-suite
that provides this high level of coverage without finding faults—MC/DC with Usage
is simply a “better” coverage criterion than the other ones we used in our experiment.
We hypothesize that MC/DC with Usage is better than the other criteria in two re-
spects. First, it seems to find more faults than any other criteria. Second, it seems
to be less sensitive to the effect of test-suite reduction. Thus, MC/DC with Usage
is the closest to the ideal coverage criterion in this domain we have seen to date; a
test-suite generated to the ideal criterion would detect all faults in the system under
test and any test-suite, large or small, providing this coverage would reveal the same
faults.

Our results are markedly different than the results reported in previous studies
[130, 81, 109]; one of the studies reports no reduction in fault finding and two studies
report a dramatic and highly varied reduction in the fault finding capability of the
reduced test-suites. In our study we observe a modest, but notable, reduction in the fault-finding capability. In our experiment, however, that reduction in fault-finding seems to be reasonably predictable; each of the five reduced test-suites we randomly generated for each coverage criterion have approximately the same fault-finding capability. This stands in stark contrast to the results in the Rothermel et al., and Jones and Harrold studies where the reduction in fault finding varied between 0% and 100% [81, 109].

We do not have a ready explanation for this phenomenon, but we speculate that it may be related to three factors; (1) the coverage criteria used in the experiment, (2) the actual coverage provided by the test-suites, and (3) the classes of faults considered. The Rothermel et al. study [109] used edge-coverage of the control flow graph (roughly equivalent to the transition coverage in our domain) and most of our criteria are more rigorous than edge-coverage. Since there seems to be a correlation between the rigor of the coverage criterion and the variability in fault-finding of the reduced test-suites, this may be part of the explanation for our results. The Jones and Harrold study [81] used MC/DC as the coverage criterion in their experiment, but the test-suites they used did not provide complete MC/DC coverage. Their reduced test-suites provided the same coverage of the code as the full suite, but the full suite did not provide coverage up to 100% of achievable coverage of the criterion of interest. In our case, we provided full coverage of every criterion. The fact that we worked from complete test-suites may have made our test-suites less susceptible to the variations if fault finding observed in their study. Finally, since we are working with languages with limited expressiveness (no iteration and dynamic memory for example), the fault classes under consideration most likely affect the results. Needless to say, further study is clearly needed to understand these issues better.

To summarize the findings, reduction of test-suite size has an unacceptable effect
on the suite’s fault finding capability. Should there be an urgent need to reduce the test-suite size because of resource limitations (in terms of, for example, time), we speculate that test-case prioritization [81] would be a better approach than test-suite reduction (or minimization). In test-case prioritization, we would not eliminate any test-cases from our test-suite; we would instead attempt to sort the test-cases based on expected fault finding potential and execute the ones deemed to be most likely to reveal faults first. We would terminate the testing when our resources are depleted. Naturally, more work is needed to determine how to prioritize test cases and also empirically evaluate if the test-case prioritization approach in fact performs better than reduced or minimized test-suites.

6.4 Threats to Validity

There are four obvious threats to the external validity that prevents us from generalizing our observations. First, and most seriously, we are using only one instance of a formal model in our experiment. Although the FGS is an ideal example—it was developed by an external industry group, it is large, it represents a real system, and is of real world importance—it is still only one instance. The characteristics of the FGS model, for example, it is entirely modelled using Boolean and enumerated variables, most certainly affects our results and makes it impossible to generalize the results to systems that, for example, contain numeric variables and constraints.

Second, we are using fault seeded models to emulate an implementation in conformance testing. As mentioned earlier, given the structure and semantics of the modelling language RSML$^e$, we assert that any reasonable implementation derived from such a model would be susceptible to the same types of faults as the model. Nevertheless, the fact that we are not using hand generated code in the experiment might bias the results and is, therefore, a concern.
Third, we are using seeded faults in our experiment. Although we took great care in selecting fault classes that represented actual faults we observed during the development of the FGS model as well as faults we deem highly likely to be introduced during implementation, fault seeding always leads to a threat to external validity—we simply do not know if our faults represent what we would see in practice.

Finally, we only considered a single fault per model. Using a single fault per specification makes it easier to control the experiment. Nevertheless, we cannot account for the more complex fault patterns that may occur in practice and the effect multiple faults may have on our ability to reveal them; for example, fault masking may affect the effectiveness of test-suites.

Although there are several threats to the external validity of our experiment, believe the results generalize to a large class of models in the critical systems domain and our results raise serious doubts about the use of any test-suite reduction techniques in this domain. Note here that test-case generation techniques that are optimized—optimized in the sense that they only generate tests needed to provide the desired coverage—would be subject to the same potential drawbacks as the test-set reduction technique we evaluated in this dissertation.

6.5 Chapter Conclusions

In this chapter, we have described an initial experiment in which we investigated the effect of test-suite reduction in the domain of automatically generated conformance test-suites. As a system-under-test, we used a model of a production sized Flight Guidance System seeded with “representative” faults. Our results confirm our two hypotheses; one can dramatically reduce the size of the automatically generated conformance test-suites while maintaining desired coverage, and the fault finding of the reduced test-suites was significantly adversely affected. Although we cannot broadly
generalize our results and further studies are needed, the experiment indicates that test-suite reduction of test-suites providing structural coverage may not be an effective means of reducing testing effort—the cost is terms of lost fault finding capability is simply too high; especially in the critical systems domain in which we are mainly interested. On a related note, the results cast doubts on the effectiveness of structural coverage criteria in general. Small (or minimal) test-suites that provide structural coverage criteria do not seem to be effective; we must understand better the reasons behind this problem and either discover ways to somehow augment the test-suites to enhance their effectiveness or define structural coverage criteria that are not sensitive to test-suite size. Our results indicate that more rigorous criteria, such as MC/DC, provide a better fault finding capability both for the full-test-suites as well as the reduced test-suites as compared to less rigorous criteria, such as variable domain and transition coverage. This hints that MC/DC is more robust to the detrimental effect of test-suite size. Nevertheless, the characteristics of coverage criteria in both the specification and implementation domain have not been well investigated and more work in this area is clearly needed.

Based on our results, we are skeptical towards any test-suite reduction techniques that aim solely to maintain structural coverage, because, in our opinion, there is an unacceptable loss in terms of test-suite quality. Thus, we advocate research into test-case prioritization techniques and experimental studies to determine if such techniques can more reliably lessen the burden of the testing effort by running a subset of an ordered test-suite, as opposed to a reduced test-suite, without little if any loss in fault finding capability.
Chapter 7

Conclusions and Future Work

In this dissertation, we have explored the utility of using specification-based structural test-coverage criteria as test-data generators in the domain of safety-critical reactive systems. We performed this evaluation through empirical investigations that compare the fault finding ability of test sets generated to various structural test-coverage criteria against randomly generated test-suites that were generated using the same effort. At the time we started our research, there were very few studies conducted that were relevant to our study and the effectiveness of test-suites automatically generated to satisfy various structural coverage criteria was not firmly established. The few limited studies that were conducted, were done on small experimental subjects using weak coverage criteria, and the results from those studies did not provide us with conclusive evidence regarding the efficacy of tests sets auto generated to satisfy structural test-coverage criteria.

Given the widespread use of structural coverage criteria such as MC/DC in safety-critical domains such as avionics systems, we conducted several empirical studies to evaluate the efficacy of tests sets auto generated to various test-coverage criteria proposed in the testing literature. We focussed on two test-coverage criteria of interest to us—MC/DC and branch coverage. We used model checking—an algorithmic technique for software verification—as the basis for automatically generating coverage adequate test sets and used a test oracle that observes internal state in addition to observable outputs.
We can draw several conclusions from the experimental results reported in this dissertation. In particular, serious issues have arisen in two areas in the context of specification-based testing. First, the coverage criteria that might be used are not well understood. Our preliminary investigations have raised serious concerns about the fault-finding ability of test-suites constructed to provide common coverage criteria. Second, the effect of test case generation tool on test effectiveness is not well understood. Our dissertation has identified some of the pitfalls encountered when using common test-coverage criteria and have identified several condition-based criteria that takes control flow into account, thereby assuring that somehow the conditions are invoked and affect the control flow. Through the expanded studies, our dissertation identifies a clear need to understand and explore the relationship between model structure, coverage criteria, test generation mechanisms, and fault finding. It would be highly desirable to have a criterion that would assure the same quality of the generated test-suites, regardless of how the model is structured. The studies conducted in this dissertation have shown that such as coverage criterion is hard to find—Although, MC/DC shows some promise to being an effective coverage criterion, even this performs poorly when the model is structured differently or when a less sensitive test oracle is used.

Another aspect of test-coverage criteria that we focussed on in this dissertation has to do with their sensitivity to test-suite reduction techniques. Our results confirm our two hypotheses; one can dramatically reduce the size of the automatically generated conformance test-suites while maintaining desired coverage, and the fault finding of the reduced test-suites was significantly adversely affected. Although we cannot broadly generalize our results and further studies are needed, the experiment indicates that test-suite reduction of test-suites providing structural coverage may not be an effective means of reducing testing effort—the cost is terms of lost fault finding
capability is simply too high; especially in the critical systems domain in which we are mainly interested. On a related note, the results cast doubts on the effectiveness of structural coverage criteria in general. Small (or minimal) test-suites that provide structural coverage criteria do not seem to be effective; we must understand better the reasons behind this problem and either discover ways to somehow augment the test-suites to enhance their effectiveness or define structural coverage criteria that are not sensitive to test-suite size. Our results indicate that more rigorous criteria, such as MC/DC, provide a better fault finding capability both for the full-test-suites as well as the reduced test-suites as compared to less rigorous criteria, such as variable domain and transition coverage. This hints that MC/DC is more robust to the detrimental effect of test-suite size.

Based on our results, we are skeptical towards any test-suite reduction techniques that aim solely to maintain structural coverage, because, in our opinion, there is an unacceptable loss in terms of test-suite quality. Thus, we advocate research into test-case prioritization techniques and experimental studies to determine if such techniques can more reliably lessen the burden of the testing effort by running a subset of an ordered test-suite, as opposed to a reduced test-suite, without little if any loss in fault finding capability. Test case prioritization techniques schedule test cases in an execution order according to some criterion. The purpose of this prioritization is to increase the likelihood that if the test cases are used in a particular order, they will more closely meet the objective than they would if they were executed in some other order. Test case prioritization can address a variety of objectives such as improving a test suite’s

In the context of our long-term objective, we believe this dissertation has gone some way towards enhancing our understanding of the effectiveness of structural test-coverage criteria for the purposes of test-data generation in the domain of safety-
critical systems. The dissertation has clearly highlighted the pitfalls one may encounter when test practitioners opt to use test-adequacy measures for test-suite generation using automated test tools. The research presented in this report has already served as a foundational work for further empirical research on test effectiveness in the domain of safety-critical systems and has fostered further work in understanding the factors that jointly influence the effectiveness of testing.

Our findings in this report are subject to at least three limitations. First, our study has focused on a relatively small number of systems but, nevertheless, we believe the systems are representative of the class of systems in which we are interested, and our results are generalizable to other systems in the avionics domain. Second, we have only used two approaches for test generation (random generation and counterexample-based). There are other methods of generating tests and these methods may yield different results. Third, we measure fault finding over seeded faults, rather than real faults encountered during development. It is possible real faults would lead to different results.

Naturally, there are many research challenges that must be overcome before we can reap the full benefits of model-based development and automated testing. This research has thrown up many questions in need of further investigation. In particular, there is a great need to rigorously evaluate existing model coverage criteria and develop new ones suitable in this domain. More specifically, we would like to know Which criteria are likely to reveal model faults; Which criteria are likely to reveal translation or implementation faults; To our knowledge, with the exception of our own investigations, there has been no comprehensive work in this area. Nevertheless, the characteristics of coverage criteria in both the specification and implementation domain have not been well investigated and more work in this area is clearly needed. Also, the results reported in this dissertation highlight the need for more research
into how coverage criterion, test generation mechanism, program structure, test oracle jointly influence test effectiveness. Moving forward, we hope that this work, will fuel further empirical research on better understanding the characteristics of desirable test-cases in the domain of safety-critical systems.
Bibliography


[73] H. Huang, W.T. Tsai, and S.Subramanian. General program slicing for software maintenance. Technical report, Department of Computer Science, University of Minnesota, 199X.


