

# Condition based coverage criteria: To use or not to use, that is the question <sup>\*</sup>

George Devaraj<sup>1</sup>, Mats P. E. Heimdahl<sup>1</sup>, and Donglin Liang<sup>1</sup>

Department of Computer Science and Engineering  
University of Minnesota, Minneapolis, MN 55455  
{devaraj,heimdahl,dliang}@cs.umn.edu

**Abstract.** When using tools to automatically generate test-suites from a specification, the selection of coverage criterion that guides the generation process is of imperative importance. In a previous study, we observed that, although a coverage criterion may seem reasonable for measuring the adequacy of a test suite, it may be unsuitable for guiding the generation of test suites; the generated test-suite may exercise only a small portion of the system under study, and thus, finds few faults. This paper presents an approach that addresses this problem. Our approach enhances existing condition coverage criteria with additional requirements that force the test generator to produce test-suites that will exercise all portions of the system. The paper also presents an empirical study that evaluates quality of the test-suite generated with the enhanced coverage criteria. The study shows that the introduction of these additional requirements can improve the fault finding capability of the generated test-suites without significant impact on the cost for generating such test-suites.

## 1 Introduction

Software testing is one of the most cost-intensive tasks in the modern software development process. The cost of effective testing is particularly acute when developing software that controls safety-critical applications in avionics, space, medical, and other control systems. Model based testing [20, 19, 11] has recently gained substantial interest in the software engineering community. Model based testing is centered around the use of a formal model of the behavior that drives testing activities such as test case generation. The resulting tests can then be used to check *conformance* between the formal model of the system and its implementation.

For systems with a finite state space, *temporal logic model checking* techniques [10, 8] provide an automatic way to generate tests for a variety of test criteria. In this approach, the test criteria are expressed as properties in temporal logic, and the counter example generating capability of model checkers is

---

<sup>\*</sup> This work has been partially supported by NASA grant NAG-1-224 and NASA contract NCC-01-001. We also want to thank the McKnight Foundation for their generous support over the years.

leveraged to obtain a test case that satisfies the test requirement encoded by the temporal property.

Previously, we demonstrated that model checking techniques can be successfully used to generate test cases from formal models representing large, industry sized systems [15]. Nevertheless, we discovered problems with the quality of test suites; the test suites were simply not effective at finding faults. The specification coverage criteria suggested in the literature that we used in our study might be adequate to *measure* the coverage of a test suite, but they were not suitable when *generating* test suites from a specification [14]. For example, the condition based coverage criterion decision coverage requires that each decision (essentially all boolean expressions) in a specification evaluate to both true and false during the execution of the tests in a test suite. The definition of the criterion does not require that the decision points have an influence on the outcome of the execution nor that the outcome of the decision is even used in the evaluation of the specification. Naturally, tests designed to reveal faults in a decision that is never invoked are unlikely to reveal faults in that condition. To our knowledge, the condition based coverage criteria required in practice (for example, in certification to DO-178B) and used in previous studies do not require us to take usage information into account. In our study, the coverage criteria we used as a basis for our test suite generation yielded test suites that did not reveal as many faults as we expected, largely because the tests generated satisfied the *letter* of the criterion—the inputs make the decision true and false—but not the *spirit* of the criterion—the decision is never actually evaluated.

To solve such problems, coverage criteria used in the generation of test suites must be rigorous enough that it makes it impossible for the generation tools to derive obviously ineffective test cases. This report investigates the use of newer coverage criteria that aim to correct the flaws we observed in [14]. We propose four new condition based criteria: single decision uses, all decision uses, MC/DC with decision uses, and Masking MC/DC with single uses. These condition based criteria include control flow information pertaining to use of specification elements (such as decisions) in their formalization. We evaluated our criteria on a real world case example of the mode logic of a flight-guidance system (FGS). The results from the experiment indicate that these revised criteria generate test suites that are better at fault detection than condition based criteria not involving control flow information.

The next section reviews related work and provides a more detailed account of the problem. In Section 3 we define new test criteria that we hope will generate more effective tests in terms of fault finding. We describe how we conducted our case study in Section 4. Section 5 analyzes the results obtained from our experiments. Finally, Section 7 discusses the implications of the results and points to future studies and experiments that are required to validate model-checking based test generation approaches further.

## 2 Previous Experiences on Model-based Test Generation

This section provides an overview of model-based test case generation using model checking and discusses the problem with the use of existing condition coverage criteria in guiding model-based test generation.

### 2.1 An Overview of Test Set Generation using Model Checkers

A formal description of the behavior of a system lends itself to be manipulated by automated means for a variety of purposes, such as, test case generation. In this context, algorithmic techniques for software verification such as model checking have been shown to be useful [1–3, 9, 18, 21, 17]. A model checker exhaustively explores the reachable state space of a given model and searches for violations of the properties under investigation [8]. Should a property violation be detected, the model checker will produce a counter-example illustrating how this violation can take place. The counter example would contain sufficient information for generating inputs that can force the system to reach a state where the violation occurs.

The counter example generation capability of a model checker can be leveraged to generate test cases according to certain test requirements. In this case, a test requirement is encoded as a property expressed in the logics used in common model checkers (e.g., the Linear Temporal Logic). We can then present the negation of this property to the model checker. If there is any test case that can satisfy the test requirement, then the model checker will find a counter example that illustrates the situation in which the negation of the property is violated (i.e., the property will hold). Therefore, the inputs generated from this counter example can satisfy the test requirement of interest. We call the negation of a property that encodes a test requirement a *trap property* [9].

The test generation approach discussed above can be used to generate test suites that satisfy a wide variety of structural coverage criteria on the formal model. In our research, we have built a framework that uses this approach to generate test suites using the model checker NuSMV [7]. Our framework can generate test cases that satisfy several existing structural coverage criteria for models specified in a state-based language called RSML<sup>-e</sup> [23]. This language falls under the same class of languages such as SCR [16] and SCADE [22]. Earlier studies [15] show that this approach to test case generation was scalable for models of industrial size complexity.

### 2.2 The Inadequacy of Existing Condition-Based Coverage Criteria

The major complexity in a safety-critical real-time application involves the various conditions that it must evaluate during the execution. In the program or the specification of such an application, more than half basic executable constructs may typically involve boolean expressions [5]. To test such a system adequately, a test suite must thoroughly exercise these boolean expressions in an attempt to ensure that no logical flaw exists in these expressions. Condition-based criteria

ranging from simple transition coverage to more complex coverage criteria such as MC/DC (Modified Condition/Decision Coverage) have been developed for characterizing the adequacy of a test suite. These criteria can be used to guide the model checker to generate test suites that are expected to thoroughly exercise the boolean expressions present in the model. In the literature, a boolean expression that appears in a conditional statement is referred to as a *decision*, and a boolean expression without any boolean operator is referred to as a *condition*.

In our scalability study [15], we observed that, for several condition-based coverage criteria, the test cases generated with our framework were typically one to two steps long. This raised some concerns as to the quality of the generated test suites in terms of fault finding. To investigate this further, we performed a pilot study [14] that evaluates the fault-finding capabilities of test suites generated with respect to various condition-based criteria proposed in literature. The pilot study was performed using a close to production model of a flight guidance system (FGS) from Rockwell Collins Inc. The study took the following steps: 1. Generating test suites using our framework for the condition based criteria; 2. Creating mutant specifications based on "naturally occurring faults"; and 3. Running the generated test suites from step 1 on the mutant specifications (generated in step 2) and recording the number of faults uncovered by each test suite.

The results from the study indicated that the common condition-based test adequacy criteria, such as the transition coverage and the decision coverage, are clearly *inadequate* for guiding the model checker to generate good test suites: the generated test suites were unable to uncover many faults in our formal model. In fact, the generated test suites perform uniformly worse in comparison to a set of random tests that were generated using the same effort (measured in terms of time). We identified two causes for this inadequacy. First, a feature of direct state setting in the flight guidance model would allow the model checker to generate test cases that technically provide the right coverage without exercising much logic in the model. This problem has been solved by introducing invariants into the model to deactivate this feature [14]. Second, the existing condition-based coverage criteria fail to account for the lazy evaluation semantics used in many existing modeling languages, including RSML<sup>e</sup> that is used in our framework.

We will illustrate the problem with the existing condition-based coverage criteria using the specification segment shown below.<sup>1</sup>

```
1: function foo():
2:     {return (b or c)}
3:
4: void main() {
5: ...
4: if (d and foo()) then ...
5: else ...
6:
```

---

<sup>1</sup> For better understanding, we use C syntax.

7:}

We are interested in finding test cases that cover the decision (b or c) embedded in `foo()`. A decision is said to be *covered* by a set of test cases if the decision can be evaluated to true with the test inputs of one test case in the set and can be evaluated false with the test inputs of another test case in the set. Such test cases are expected to exercise the decision and to uncover faults in this decision. According to the definition, the two test input vectors  $(b, \neg c, \neg d)$  and  $(\neg b, \neg c, \neg d)$  will achieve decision coverage of the decision on line 2. However, if the language has lazy-evaluation, `foo()` will not get called and the decision (b or c) will not even get evaluated. Therefore, these tests will not uncover any faults embedded in the decision. The test sets generated for other condition-based coverage criteria also suffer similar problems. Additional requirements may need to be introduced in the definitions of these criteria so that the model checker can generate test cases that guarantee to exercise the constructs of interests in an intended way.

### 3 Newer Coverage Criteria

In order to remedy the problem with condition based criteria discussed above, we need to strengthen their definitions to require that the required test execution path is actually exercised. To guarantee that the test execution path is actually taken, we define what we call an "evaluation context".

#### 3.1 Evaluation Context

**Definition 1.** *An evaluation context is defined as a constraint that must be satisfied so that a construct of interest (referenced in some part of the program/specification) is exercised during test execution.*

For instance, in the example discussed earlier in section 2, the condition "d" evaluating to false would be the evaluation context for the decision on line 2 (b or c).

We surmise that condition based coverage criteria used in our pilot study failed to detect faults embedded in the specification because the notion of "evaluation context" was missing in their definitions. The resulting test cases therefore were ineffective in uncovering faults, because the required test execution path was never taken on test case execution. The solution then is to come up with stronger definition for these condition based test coverage criteria which incorporates this notion of "evaluation context". In the rest of this paper, we refer to these newer condition based criteria as "augmented criteria", since their definitions are augmented with this notion of "evaluation context". The following subsections describe how one would go about determining evaluation contexts for constructs of interest (such as decisions) in formal models and later sections define the augmented versions for these condition based criteria.

### 3.2 Determining Evaluation contexts

Figure 1 shows the framework for test set generation using augmented test coverage criteria. We are targeting the input language of the model checker NuSMV from our source language RSML<sup>-e</sup>. Trap properties for test criteria such as transition, decision coverage are auto generated from the formal model expressed in RSML<sup>-e</sup>. The translated NuSMV model and the Trap properties are used to drive test case generation for each respective test coverage criteria. The critical piece we are adding here to the framework is the evaluation context encoder which performs the following two functionalities:

- a. Determine Evaluation contexts for constructs of interest and encode those in the formal model of the target language (In our case the input language of the model checker NuSMV)
- b. Encode information regarding evaluation context in the formalization of the test coverage criteria. Since, we use trap properties to derive test set generation, this requires us to add the information in the LTL or CTL temporal logic formula, as the case may be.

For illustrative purposes, let us modify the example discussed in section 2 to incorporate two references to the decision (b or c) encapsulated by the boolean function foo()(lines 4 and 9). Note that we use C notation to discuss the examples, for clarity purposes. The same issue is inherent in formal specification languages such as RSML<sup>-e</sup>, SCR etc. that support lazy-evaluation of boolean expressions.

```
1: function foo():
2:     {return (b or c)}
3:
4: void main() {
5: ...
4: if (d and foo()) then ...
5: else ...
6:
7:
8: if (a and b) then ...
9: else if (foo() and c) then ...
10: else ...
11:
12:
13:}
```

The evaluation contexts for the decision encapsulated by foo() can be encoded in the input language of the model checker NuSMV as follows:

```
init(foo_usageflag) := 0;
```

```

next(foo_usageflag) :=
  case (!(next(d)): 1;
  case (!(next(d) & next(foo.result)) & !(next(a)
    &(next(b)))):1;
  1: 0;
  esac;

```

The first case determines the evaluation context for `foo()` which is referenced in line 4. The second case determines the evaluation context for `foo()` referenced in line 9. Note however that in this case, since `foo()` is a boolean function with no parameters, we don't require to encode both these evaluation contexts for `foo()` in the model. Having one valid evaluation context will suffice, since it will guarantee that the decision `b` or `c` is exercised for both possible outcomes. But in the case of functions with parameters, we would require to capture all valid evaluation contexts in the formal model.

Once the evaluation context had been captured in the formal model, the associated trap properties that will result in test cases for decision coverage of `b` or `c` can be formalized in LTL as follows:

1.  $G(! (X(\text{foo.result} = 1) \ \& \ X(\text{foo\_usageflag} = 1)))$
2.  $G(! (X(\text{foo.result} = 0) \ \& \ X(\text{foo\_usageflag} = 1)))$

The trap properties explicitly encode the requirement that the decision encapsulated by `foo()` should be invoked. The resulting counter example (or test case) will contain the test inputs necessary for satisfying the evaluation context for `foo()`. The following subsection defines some of the augmented coverage criteria that incorporate aspects from condition and control in their definition (which is captured by the notion of evaluation contexts).

### 3.3 Augmented Test Coverage Criteria Definitions

In each of these definitions, 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. The sequence of inputs will guide the model 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. Note that for the condition based coverage criteria, a *condition* is defined as a Boolean expression that contains no Boolean operators and a *decision* is Boolean expression consisting of conditions and zero or more Boolean operators.

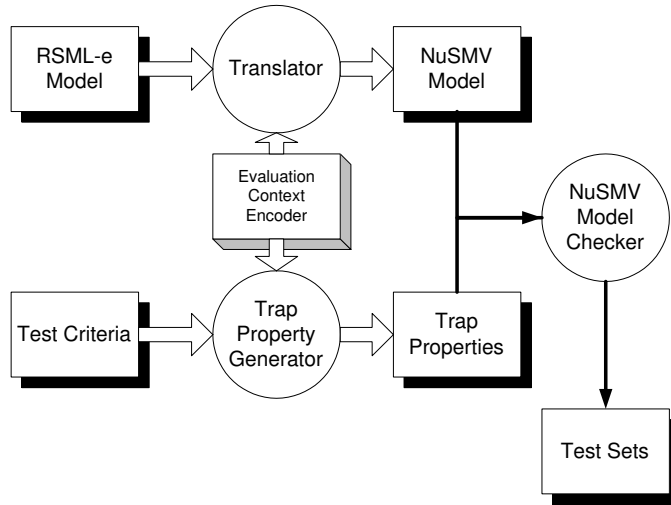


Fig. 1. Generation of Test Sets incorporating Evaluation Contexts.

### Single decision uses

**Definition 2.** A test suite is said to achieve single decision use coverage of a given state based specification, if each decision has a valid evaluation context and 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.

### All decision uses

**Definition 3.** A test suite is said to achieve all decision uses coverage of a given state based specification, if every instance of a decision has a valid evaluation context and 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.

**MC/DC with decision uses** MC/DC coverage with decision uses is a variant of the code based coverage criterion called modified condition/decision coverage. (MC/DC) was developed to meet the need for extensive testing of complex boolean expressions in safety-critical applications [4]. MC/DC requires us to show the following:

- Every condition (clause) within the decision has taken on all possible outcomes at least once
- Every condition has been shown to independently affect the outcome of the decision

**Definition 4.** A test suite is said to achieve MC/DC coverage with decision uses of a given  $RSML^e$  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 has a valid evaluation context.*

**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 [12]. 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 of a condition on a decision's outcome.

The test sets generated to these augmented coverage criteria will ensure that the evaluation context is valid on test set execution, thereby guaranteeing that the proper part of the specification is exercised. In essence, what we end up with are test cases that satisfy the spirit of the criterion. One would expect that these would perform better than those test cases that do not incorporate the notion of "evaluation context". In order to confirm this, we performed a case study to evaluate the efficacy of test sets generated to these augmented test coverage criteria. The following section briefly discusses the case example and the experimental setup used in this study.

## 4 Evaluating Augmented test coverage criteria

An empirical study was conducted in order to determine the feasibility and usefulness of these augmented criteria. The goal of this empirical study was to evaluate these augmented criteria in terms of their fault finding capability and compare it with the other representative condition based criteria that don't use this notion of evaluation context in their definitions. A knowledge of the practical trade-offs involved in generating tests using these criteria will help us in identifying a set of strong coverage criteria that are most suitable in the domain of model based testing.

### 4.1 Hypothesis

The augmented coverage criteria discussed earlier guarantee that the proper part of the specification is exercised upon test set execution. As such, we can surmise that they will be better at finding more faults than coverage criteria that are purely condition based. We therefore develop the following hypothesis about augmented test coverage criteria:

**Hypothesis 1:** Test sets that are derived from augmented coverage criteria perform better in terms of fault finding than purely condition based coverage criteria.

The following paragraphs discuss the case example used in our study and the experimental setup.

## 4.2 Case Example: The Flight Guidance System

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<sup>2</sup>. 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 aircraft's current and desired state and compute the pitch and roll guidance commands. In this case study we have used the mode logic.

The primary modes of interest in the FGS are the 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 is ideally suited for test case generation using model checkers since it is discrete—the mode logic consists entirely of enumerated and Boolean variables.

The experiment involved the following steps:

1. **Test-suite generation:** We used the original FGS specification expressed in RSML<sup>-e</sup> to generate a test suite for a coverage criterion of interest, for example, decision coverage, and measured the effort involved. We used the model checker NuSMV as the test generation engine.
2. **Fault Injection:** We generated 100 faulty specifications of the FGS by randomly seeding one fault per faulty specification. In creating the 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. This was done to identify "naturally occurring" or "representative" faults. We identified four different fault classes for the RSML<sup>-e</sup> language—variable replacement (VR), condition insertion (CI), condition negation (CN), condition replacement (CR). These faults mainly involved syntactic changes, for instance, in a variable replacement faults, the variable reference is replaced with a reference to another variable of the same type.

---

<sup>2</sup> We thank Dr. Steve Miller and Dr. Alan Tribble of Rockwell Collins Inc. for the information on flight control systems and for letting us use the RSML<sup>-e</sup> models they have developed using NIMBUS.

3. **Test execution on the faulty specifications and fault detection:** We ran the full test suite on the 100 faulty specifications and recorded the number of faults uncovered.

The next section presents the results obtained from our study and provides a detailed analysis on the efficacy and feasibility of these augmented coverage criterion in the domain of formal specifications.

## 5 Experimental Results and Analysis

The results from our experiment are presented in Table 1. As a baseline for comparisons, we generated a random test suite using the same effort (measured in time) we used in generating tests for the various test coverage criteria discussed in this paper. We ran these random tests against the faulty specifications and recorded the number of faults detected.

Table 1 presents the fault finding ability (classified per fault type) of the several test suites generated for both condition based as well as the augmented criteria. It is worth noting that during the experiment, coverage criteria such as all decision uses was found to be intractable for test case generation purposes using the model checking approach. This can be attributed to the exponential increase in the state space of the model brought about by the introduction of extra boolean variables that are used to track multiple evaluation contexts for a decision. Therefore, we do not report any results for this coverage criterion.

The results from Table 1 supports the hypothesis that test cases generated using augmented test coverage criteria perform better at fault finding than test cases generated to the purely condition based criteria considered in this study. Also all the augmented test coverage criteria perform significantly well in comparison with random testing. Strong criteria such as MC/DC with decision uses and masking MC/DC with decision uses detect the largest number of faults. Also, the number of faults that were uncovered by these two coverage criteria reached the theoretical maximum (72). This is because our fault seeder does not distinguish between semantically equivalent mutants. So in essence, there were 72 semantically different mutants that were generated by the fault seeder. Therefore, augmented test coverage criteria such as MC/DC with uses and Masking MC/DC with uses close to being an ideal coverage criteria in this model based domain because they detect all types of faults seeded in the study. In another related study [13], it was found that these two coverage criteria are least sensitive to the effects of test conditions such as test set reduction. These results basically provide strong evidence concerning the robustness of these two coverage criteria.

One can also observe that the test suites generated for these two criteria are exactly equivalent in terms of fault detection. This is in agreement with the result in [6], where the Masking form of the MC/DC coverage criterion was found to provide equivalent error detection in comparison to the strict form of the MC/DC.

Another important point worth noting is that some of the fault types are more difficult to detect by the test suites than others. In our case study, it was

found that a lot of the condition insertion faults (CI) go undetected. This is because changing a "C" to a "T" or a "F" may *not* always change the behavior of the specification. So, in such a scenario the test cases will not report this as a fault.

Some of the implications that can be derived from this case study are that these augmented criteria ensure correct formalization of the test criteria in the model based testing domain. They do this by requiring that specification elements must get invoked or used. This makes the coverage criteria sensitive to the semantics of the specification language used and the test generation algorithm (model checking). As such, the resulting test cases are more effective and detect significantly more faults than other condition based criteria. Weaker criteria such as state, transition are clearly inadequate for test case generation purposes because of their poor fault detecting potential.

Test Criteria	VR	CN	CI	CR	Total
Random	21	25	5	15	66
Variable Domain	14	15	2	4	32
Transition	20	24	5	15	64
Decision	23	24	5	15	67
Decision Usage	23	24	7	15	69
Masking MC/DC	22	25	7	16	70
Masking MC/DC Usage	23	25	8	16	72
MC/DC	22	25	7	16	70
MC/DC Usage	23	25	8	16	72

**Table 1.** Fault detection capability of Test set generation for various criteria along with their fault detection capability

Test Criteria	Size	Time (s)	Memory (MB)
Variable Domain	100	81.61	58
Transition	313	193.67	80
Decision	435	511	95
Decision Usage	478	2615	138
Masking MC/DC	552	8651	154
Masking MC/DC Usage	361	8427	158
MC/DC	537	8796	162
MC/DC Usage	334	8542	163

**Table 2.** Test set generation times and memory usage in NuSMV for various criteria

Another aspect that needs to be considered regarding the use of these augmented criteria is whether it is tractable to use them for test generation purposes using model checking techniques. As noted in Table 2, the more rigorous the coverage criteria, the more expensive it is to generate test cases that satisfy them. Also, counter example generation times for augmented coverage criteria is slightly more expensive when compared to purely condition based coverage criteria. For some of the stronger coverage criteria like MC/DC(both with and without decision uses) the cost of counter example generation using NuSMV is high. As earlier explained in section 3.3, this is due to the complexity of the MC/DC criterion which requires us to capture constraints on the value on atomic conditions over two test sequences. As a result, the trap properties corresponding to a MC/DC requirement are often long and complex. The complexity of the trap property in turn affects the counter example generation time using a bounded model checker, since the performance of the LTL model checker is exponential in the size of the formula.

## 6 Threats to validity

There are three obvious threats to the validity in our study 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 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.

Although there are several threats to the external validity of our experiment, we believe the results are representative of a large class of models in the critical systems domain.

## 7 Summary and Conclusions

In summary, we have identified a set of test coverage criteria based on the information of condition and control flow (augmented criteria) that are suitable for test case generation purposes in the domain of formal specifications. An empirical study was undertaken to evaluate the effectiveness of these criteria in terms of fault finding. The empirical study was performed on a production sized flight

guidance system model seeded with "representative" faults. The results from the study confirm our hypothesis that: augmented criteria perform better at finding varied number of faults in comparison to random tests and other test criteria that are based purely condition based.

Furthermore, our results indicate that more rigorous criteria, such as MC/DC, provide a better fault finding capability as compared to less rigorous criteria, such as variable domain and transition coverage.

Although we cannot broadly generalize our results and further studies are needed to determine the robustness of these criteria, the experiment indicates that augmented test coverage criteria show promise as an means to generate effective test cases in the critical systems domain.

## References

1. *Proceedings of The First International Workshop on Automated Program Analysis, Testing and Verificaiton, ICSE 2000*. 2000.
2. Paul E. Ammann and Paul E. Black. A specification-based coverage metric to evaluate test sets. In *Proceedings of the Fourth IEEE International Symposium on High-Assurance Systems Engineering*. IEEE Computer Society, November 1999.
3. Paul E. Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. In *Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 46–54. IEEE Computer Society, November 1998.
4. J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, pages 193–200, September 1994.
5. J.J. Chilenski and S.P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9:193–200, September 1994.
6. John Chilenski. An investigation of three forms of the modified condition decision coverage (mcdc) criterion. Technical Report DOT/FAA/AR-01/18, Office of Aviation Research, Washington, D.C., April 2001.
7. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A New Symbolic Model Checker, 2004. Available at <http://nusmv.irst.itc.it/>.
8. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.
9. Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. *Software Engineering Notes*, 24(6):146–162, November 1999.
10. O. Grumberg and D.E.Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.
11. I.J. Hayes. Specification direced module testing. In *IEEE transactions on Software Engineering*, January 1986.
12. K.J. Hayhurst, D.S. Veerhusen, and L.K. Rierson. A practical tutorial on modified condition/decision coverage. Technical Report NASA/TM-2001-210876, NASA, 2001.
13. Mats P.E. Heimdahl and George Devaraj. Test-suite reduction for model-based tests: Effects on test quality and implications for testing. In *Proceedings of the 19th*

- International Conference on Automated Software Engineering (ASE '04)*, Linz, Austria, September 2004.
14. Mats P.E. Heimdahl, George Devaraj, and Robert J. Weber. Specification test coverage adequacy criteria = specification test generation inadequacy criteria? In *Proceedings of the Eighth IEEE International Symposium on High Assurance Systems Engineering (HASE)*, Tampa, Florida, March 2004.
  15. Mats P.E. Heimdahl, S. Rayadurgam, Willem Visser, George Devaraj, and Jimin Gao. Auto-generating test sequences using model checkers: A case study. In *3rd International Workshop on Formal Approaches to Testing of Software (FATES 2003)*, 2003.
  16. C. L. Heitmeyer, B. L. Labaw, and D. Kiskis. Consistency checking of SCR-style requirements specifications. In *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, March 1995.
  17. Hyoung Seok Hong, Sung Deok Cha, Insup Lee, Oleg Sokolsky, and Hasan Ural. Data flow testing as model checking. In *Proceedings of the International Conference on Software Engineering*, Portland, Oregon, May 2003.
  18. Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A temporal logic based theory of test coverage and generation. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '02)*, Grenoble, France, April 2002.
  19. J.Dick and A. Faivre. Automating the generation and sequencing of test cases from model based specifications. In *Proceedings of FME '93:Industrial-Strength Formal Methods*, Odense, Denmark, 1993.
  20. A. Jefferson Offutt, Yiwie Xiong, and Shaoying Liu. Criteria for generating specification-based tests. In *Proceedings of the Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, October 1999.
  21. Sanjai Rayadurgam and Mats P.E. Heimdahl. Coverage based test-case generation using model checkers. In *Proceedings of the 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2001)*, pages 83–91. IEEE Computer Society, April 2001.
  22. Esterel Technologies. Scade suite product description. <http://www.esterel-technologies.com/v2/scadeSuiteForSafetyCriticalSoftwareDevelopment/index.html>, 2004.
  23. Jeffrey M. Thompson, Mats P.E. Heimdahl, and Steven P. Miller. Specification based prototyping for embedded systems. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, number 1687 in LNCS, pages 163–179, September 1999.