Specification Test Coverage Adequacy Criteria = Specification Test Generation *In*adequacy Criteria?

Mats P.E. Heimdahl, Devaraj George, Robert Weber

Department of Computer Science and Engineering, University of Minnesota
E-mail: {heimdahl,devaraj,weber}@cs.umn.edu

Abstract

The successful analysis technique model checking can be employed as a test-case generation technique to generate tests from formal models. When using a model checker for test case generation, we leverage the witness (or counter-example) generation capability of model-checkers for constructing test cases. Test criteria are expressed as temporal properties and the witness traces generated for these properties are instantiated to create complete test sequences, satisfying the criteria. In this report we describe an experiment where we investigate the fault finding capability of test suites generated to provide three specification coverage metrics proposed in the literature (state , transition, and decision coverage). Our findings indicate that although the coverage may seem reasonable to measure *the adequacy of a test suite, they are unsuitable when used to* generate *test suites. In short, the generated test sequences technically provide adequate coverage, but do so in a way that tests only a small portion of the formal model. We conclude that automated testing techniques must be pursued with great caution and that new coverage criteria targeting formal specifications are needed.*

1. Introduction

Model checking techniques have been proposed as one method for automatically deriving test sequences from formal models or, in certain circumstances, from code [4, 7, 3, 8, 16, 13]. These proposed test case generation approaches leverage the witness (or counter-example) generation capability of model-checkers for constructing test cases. Test criteria are expressed as temporal properties. Witness traces generated for these properties are instantiated to create complete test sequences satisfying the criteria. Nevertheless, one of the issues that often stymies model-checking is the state-space explosion problem. As the size of the state-space to be explored increases, model-checking might become too time-consuming or infeasible. But, in the context of test generation based on structural properties one is interested in falsifying properties so that counter-examples can be instantiated to test sequences. In a previous study, we demonstrated that finding violations of the properties characterizing a test case is relatively easy and that the counter-examples can be constructed easily even for quite large models [11]. Nevertheless, our experiences from this case study raised some concerns regarding the *quality* of the test cases generated. In particular, the test cases were all very short—typically one step long—and, although they provided the specification coverage sought, raised concerns as to their effectiveness in revealing faults. To evaluate the quality of the generated tests we conducted a large case study comparing the fault-finding capabilities of tests automatically generated with a model checker to tests generated randomly. To our surprise (and dismay) the randomly generated tests performed better than the generated structural tests. In this paper we report on our case study, discuss the reasons for the poor performance of the structural tests, and point out the need for new coverage criteria suitable for test-case generation from formal specifications.

To evaluate the fault finding capability of test suites generated to provide three common specification test adequacy criteria (state, transition, and decision coverage), we conducted an experiment using a model of the mode-logic in a production sized flight-guidance system, written in the RSML^{-e} language [19, 20]. We used our framework for specification-based test generation using the NuSMV model-checker [11, 15]. The generated tests were then executed on versions of the specification with seeded faults. The purpose of this study was to determine how well the structural tests were able to reveal faults as compared to the same effort expended on random testing.

To summarize our findings, our experiment indicates that the common specification test adequacy criteria we evaluated are woefully *inadequate* and are not likely to reveal faults in our formal model. We identify two reasons for this inadequacy: (1) the structure of the flight guidance model leads the model checker to find test cases that technically provide the right coverage but do not exercise the logic of the model, the (2) the semantics of the RSML^{-e} specification language makes some coverage criteria unsuitable. We believe theses findings generalize to other systems and formalisms and, therefore, new specification coverage criteria must be developed.

The rest of the paper is organized as follows. Section 2 provides a short overview of related efforts in the area of test-generation using model checking techniques and briefly describes our overall approach. Section 3 describes our testing framework and the formal specification language on which we based our work. We describe how we conducted our experiment in Section 4. Section 5 briefly discuss the test coverage criteria used for this study. Sections 6 and 7 analyzes the results obtained from our experiments, and Section 8 discusses the implications of the results and points to further research opportunities.

2. Finding Tests with a Model Checker

Model checkers build a finite state transition system and exhaustively explore the reachable state space searching for violations of the properties under investigation [6]. Should a property violation be detected, the model checker will produce a counter-example illustrating how this violation can take place. In short, a counter-example is a sequence of inputs that will take the finite state model from its initial state to a state where the violation occurs.

A model checker can be used to find test cases by formulating a test criterion as a verification condition for the model checker. For example, we may want to test a transition (guarded with condition C) between states A and B in the formal model. We can formulate a condition describing a test case testing this transition—the sequence of inputs must take the model to state A; in state A, C must be true, and the next state must be B. This is a property expressible in the logics used in common model checkers, for example, the logic LTL. We can now challenge the model checker to find a way of getting to such a state by negating the property (saying that we assert that there is no such input sequence) and start verification. We call such a property a *trap property* [8]. The model checker will now search for a counterexample demonstrating that this trap property is, in fact, satisfiable; such a counterexample constitutes a test case that will exercise the transition of interest. By repeating this process for each transition in the formal model, we use the model checker to automatically derive test sequences that will give us transition coverage of the model. Obviously, this general approach can be used to generate tests for a wide variety of structural coverage criteria, such as all state variables have take on every value, and all decisions in the model have evaluated to both true and false. The test generation process is outlined in Figure 1.

The approach discussed above is not unique to our group, several research groups are actively pursuing model checking techniques as a means for test case generation [1, 2, 3, 8, 13, 16, 12]. Nevertheless, to our knowledge, no experimental data is available about the fault finding capability of test suites automatically generated to various specification coverage criteria.

3. NIMBUS and RSML^{-e}

Figure 2 shows an overview of the NIMBUS tools framework we have used as a basis for our test case generation engine. The user builds a behavioral model of the system in the fully formal and executable specification language RSML^{-e} (see below). After evaluating the functionality and behavioral correctness of the specification using the NIMBUS simulator, users can translate the specifications to the PVS or NuSMV input languages for verification (or test case generation as is the case in this report). The set of LTL trap properties required to use NuSMV to generate test sequences are obtained by traversing the abstract syntax tree in NIMBUS and then outputting sets of properties whose counterexamples will provide the correct coverage (the coverage criteria and associated properties are discussed in the next section).

To generate test cases in NIMBUS, the user would invoke the following steps. First, the user would automatically translate the model to the input language of NuSMV and automatically generate the set of trap properties of interest to achieve desired coverage. Second, the trap properties are automatically merged with the NuSMV model and the NuSMV model checker is invoked to collect the counterexamples. Third, the counterexamples are processed to extract test sequences in a generic intermediate test representation. The intermediate test representation contains (1) the input in each step, (2) the expected state changes (to state variables internal to the RSML^{-e} model), and (3) the expected outputs (if any). Finally, the intermediate test representation would be translated to the input format for whatever testing tool is used to test the system under test.

The NIMBUS tools discussed above all operate on the RSML^{-e} notation — RSML^{-e} is based on the Statecharts [9] like language Requirements State Machine Language (RSML) [14]. RSML^{-e} is a fully formal and synchronous data-flow language without any internal broadcast events (the absence of events is indicated by the $^{-e}$).

An RSML^{-e} 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

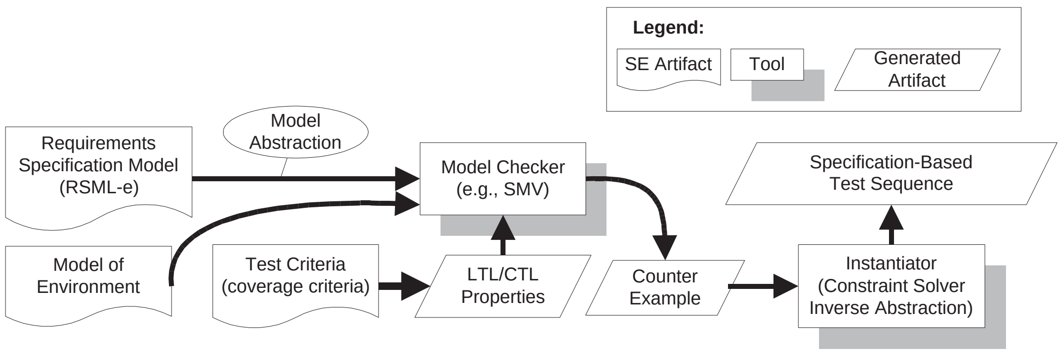
Legend:
SE Artifact
Tool
Generated Artifact
Requirements Specification Model (RSML-e)
Model Abstraction
Model Checker (e.g., SMV)
Specification-Based Test Sequence
Model of Environment
Test Criteria (coverage criteria)
LTL/CTL Properties
Counter Example
Instantiator (Constraint Solver Inverse Abstraction)

Figure 1. Test sequence generation overview and architecture.

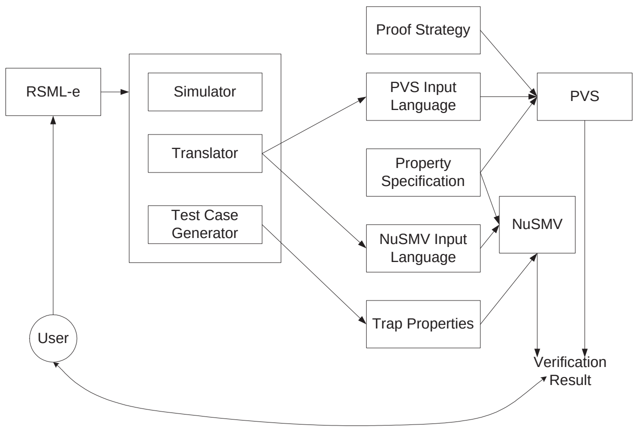

Figure 2. Verification Framework.

increased readability and ease of use.

Figure 3 shows a specification fragment of an RSML^{-e} specification of the Flight Guidance System[1]. The figure shows the definition of a state variable, ROLL. ROLL is the default lateral mode in the FGS mode logic.

The conditions under which the state variable changes value are defined in the TRANSITION clauses in the definition. The condition tables are encoded in the macros, Select_ROLL and Deselect_ROLL. The tables are adopted from the original RSML notation—each column of truth values represents a conjunction of the propositions in the leftmost column (F represents the negation of the proposition and a '*' represents a "don't care" condition). If a table contains several columns, we take the disjunction of the columns; thus, the table is a way of expressing conditions in a disjunctive normal form.

[1]We use here the ASCII version of RSML^{-e} since it is much more compact than the more readable typeset version.

```
STATE_VARIABLE ROLL : Base_State
   PARENT          : Modes.On
   INITIAL_VALUE   : UNDEFINED
   CLASSIFICATION  : State

   TRANSITION UNDEFINED TO Cleared IF NOT Select_ROLL()
   TRANSITION UNDEFINED TO Selected IF Select_ROLL()
   TRANSITION Cleared TO Selected IF Select_ROLL()
   TRANSITION Selected TO Cleared IF Deselect_ROLL()

END STATE_VARIABLE


MACRO Select_ROLL() :
   TABLE
      Is_No_Nonbasic_Lateral_Mode_Active()    : T;
      Modes = On                              : T;
   END TABLE
END MACRO

MACRO Deselect_ROLL() :
   TABLE
      When_Nonbasic_Lateral_Mode_Activated()  : T *;
      When(Modes = Off)                       : * T;
    END TABLE
END MACRO
```

Figure 3. A small portion of the FGS specification in RSML^{-e}.

4. Experiment Overview

In our experiment, we were interested in answering one 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 coverage criteria, state, transition, and decision coverage. (The criteria will be discussed in detail in Section 5.)

To provide realistic results, we conducted the experiment using a close to production model of a flight guidance sys-

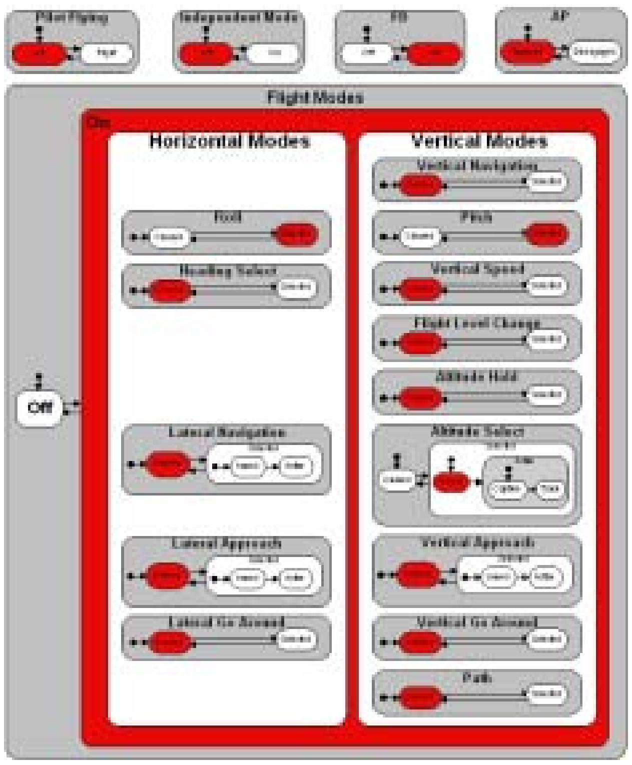

Figure 4. Flight Guidance System

tem from Rockwell Collins Inc.[2] 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 aircraft's current and desired state and compute the pitch and roll guidance commands. In this case study we have used the mode logic.

Figure 4 illustrates a graphical view of a FGS in the NIMBUS environment. 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.

[2]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^{-e} models they have developed using NIMBUS.

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

Figure 5. An example fault seeded into the FGS model.

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 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.

We used our fault seeder to generate 100 faulty specifications (25 for each fault class). As an example, Figure 5 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."

We then performed the experiment by conducting the following steps for several different structural coverage criteria:

1. We used the original specification to generate a test suite to a coverage criterion of interest, for example, transition coverage, and measured the effort involved.
2. We ran the test suite on the 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.

In the remainder of this paper we provide a detailed description activities involved in the case study and discuss our findings.

5. Coverage Criteria

For the case study described in this report, we have selected to use three representative specification coverage criteria; state coverage, transition coverage, and decision coverage.

In the following discussion, a *test case* is to be understood as a sequence of values for the input variables in an RSML^{-e} specification. This sequence of inputs will guide the RSML^{-e} specification from its initial state to the structural element, for example, a transition, the test case was designed to cover. A *test suite* is simply a set of such test cases. As we briefly explained, trap properties are used to generate counter-examples using a model checker. These properties are derived from the structural coverage criteria. For the purposes of illustration, we use the FGS example discussed in Section 4.

State coverage:

Definition 1. *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.*

Consider, for example, the state variable `ROLL` in the FGS specification example:

```
STATE_VARIABLE ROLL : { Cleared, Selected, UNDEFINED };
```

A test suite would achieve state coverage on `ROLL`, if for each of its three different possible values, there is a test case in which `ROLL` takes that value. Note that a single test case might actually achieve this coverage by assigning different values to `ROLL` at different points in the sequence. To provide a comprehensive test suite, however, in this case study we generate one test case for each state variable value. One could use the following LTL formulas to generate the test cases:

```
1. G ~(ROLL = Cleared)
2. G ~(ROLL = Selected)
3. G ~(ROLL = UNDEFINED)
```

In each case, the property asserts that `ROLL` can never have a specific value and the counter-example produced is a sequence of values for the system variables starting from an initial state and ending in a state where `ROLL` has the specific value.

Transition Coverage:

Definition 2. *A test suite is said to achieve transition coverage of a given RSML^{-e} specification, if each guard condition on a transition (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.*

As an example, consider the transition defined for the `ROLL` state variable in Figure 3. If we consider the transitions to `Selected` guarded by the condition encapsulated in the macro `Select_ROLL()`, test cases to provide decision coverage of this decision can be generated using the following two trap properties.

```
1. G((Select_ROLL()) -> ~(ROLL = Selected))
2. G(~(Select_ROLL()) -> (ROLL = Selected))
```

Decision Coverage:
In our discussion of decision coverage, we have based our definitions on our understanding of how the Federal Aviation Administration (FAA) views decision coverage in its guidelines for software development in accordance with the DO-178B standard for critical airborne software [10, 17, 18].

Assuming the Boolean operators AND, OR, and NOT, we use the following definitions for condition and decision:

Condition: A Condition is defined as a Boolean Expression containing no Boolean Operators. For instance, a Boolean variable or a relational expression would be considered as a condition.

Decision: A Decision is Boolean Expression consisting of conditions and zero or more Boolean operators.

Definition 3. *A test suite is said to achieve decision coverage of an RSML^{-e} specification, if every decision appearing anywhere in the specification 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.*

Note here that we are interested in all decisions, not only decisions in traditional *decision points* (such as branches). Consider the code example below.

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

In this code fragment we have two decisions (statements 1 and 3) and we have to make sure the test suite makes both of them true and false in some test to achieve decision coverage. Note here that it is not required that the truth value of

A must have an impact on the decision point on statement 3, that is, the two test inputs $(b, \neg c, \neg d)$ and $(\neg b, \neg c, \neg d)$ would achieve decision coverage of the decision on statement 1, but the value of A would not have an an impact on the outcome of the decision point on statement 3.

If we adopt this definition of decision coverage to the specification domain, every Boolean expression used to guard a transition, used to guard interactions with the environment, encapsulated in a macro, and used to guard the cases in a case statement (used in function definitions) will be considered decisions.

To illustrate decision coverage in the context of RSML^{-e} consider the state variable definition below (where we have abstracted away the actual clauses for space reasons).

```
 STATE_VARIABLE Onside_FD: On_Off
  PARENT: None

  EQUALS On IF TABLE
               clause1: T *;
               clause2: * F;
             END TABLE

  EQUALS Off IF
            TABLE
              clause1:  T F *;
              macro3(): * T *;
              clause3:  * * F;
            END TABLE
END STATE_VARIABLE

MACRO macro3()
    clause21: T *;
    clause23: * F;
END MACRO
```

Here we would have to cover the decisions in the transition definition of the `Onside_FD` variable as well as the macro to which one of these transitions refers. The trap properties below will provide decision coverage on the transitions for `Onside_FD`.

```
1.G(!(X(clause1) V !X(clause2)))
2.G(X(clause1) V !X(clause2))
3.G(!(X(clause1 V
     (!clause1 & m_macro3.result) V
     !clause3))
4.G((X((clause1 V
    (!clause1 & m_macro3.result) V
    !clause3))
```

The decision in the macro is addressed with two additional trap properties.

```
1.  G(! X(m_macro3.result = 1))
2.  G(! X(m_macro3.result = 0))
```

State	Transition	Decision
246	342	424

Table 1. Number of trap properties generated for each coverage criterion

State	Transition	Decision
1.000	1.127	1.125

Table 2. Average length of the test cases in the test suites

Test suites generated to these trap properties will guarantee that each decision would take on both the value true and false.

To summarize, we have automated the generation of trap properties for a collection of structural coverage criteria of formal specifications. In this case study we are using the three representative criteria described above; state coverage, transition coverage , and decision coverage.

6. Experimental Results

As mentioned in Section 4, we generated tests to provide the coverage discussed in the previous section. Table 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 [11]. From our previous study, of interest for this paper is the average length of the test cases generated (Table 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 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 3. The results here only reflect the relative fault finding capability—we made no effort to determine how

	Rand.	State	Tran.	Dec.
Var. Repl.	21	14	20	20
Cond. Ins.	5	2	3	4
Cond. Rem.	15	4	11	8
Cond. Neg.	25	12	24	24
Total	66	32	58	56

Table 3. Number of faults revealed by each test suite

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.

7. Discussion

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

7.1 Model Structure

As mentioned above, the test-case length was universally very short and, seemingly, quite poor at revealing faults. To explain this phenomenon, let us take a closer look at the flight guidance system used as a case-example.

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

Most test-cases that the model checker found took advantage of this particular feature of the FGS model—the test cases made the FGS under test the inactive FGS and simply used the other FGS interface to drive the model to the desired state (or take the desired transition, or make the desired condition true or false). For example, when the FGS is active there are some rather complex rules for when to enter the `ROLL` mode. When the FGS is inactive, on the other hand, all that is required to enter `ROLL` mode is to command it there with an input variable. Thus, is is possible for test cases to achieve coverage by *commanding* the FGS to go to states and take transitions—they do not exercise the *actual mode logic* of the FGS. Naturally, such test cases will not reveal any faults seeded in the mode logic of the FGS. Unfortunately, this is exactly the test case that the bounded model checker is likely to find—it is most often the simplest possible way of achieving the test objective.

State	Transition	Decision
1.114	1.131	1.139

Table 4. Average length of the test cases in the test suites after using the invariant

To solve this problem, we can simply prohibit messages on the FGS to FGS interface and in that way force the model checker to find test cases that actually use the mode logic. For example, we can add an invariant to the model checker prohibiting the FGS from being inactive.

```
INVAR (Is_This_Side_Active = 1)
```

This is technically quite an easy thing to do, but it requires intimate knowledge of the existence of the FGS-FGS interface and the knowledge that this interface can be "abused" by the test case generation automation to achieve its objectives. We see this as an issue in many critical systems since they are typically designed with one or more hot-spares that need to be kept synchronized.

To investigate the how severe this effect was on the fault finding capabilities of our test suites, we regenerated the suites with the above mentioned invariant that keeps the FGS model in the active state. As can be seen in Tables 4 and 5, the test case length as well as the fault finding capability of the test suites went up. The performance is still inadequate, however, and random testing performs better. From this work we have come to the conclusion that state and transition coverage are clearly inadequate in this domain—more elaborate coverage is necessary. We have some hope for various condition based coverage, for example, modified decision and condition coverage (MC/DC) [5], but our experiences with simple decision coverage (discussed next) raises some issues with the adoption of condition based coverage criteria in the specification domain.

7.2 Inadequate Coverage Criteria

To illustrate the problems with condition based coverage criteria, consider again the small code fragment we used

	State	State(a)	Tran.	Tran.(a)
Var. Repl.	14	17	20	21
Cond. Ins.	2	1	3	3
Cond. Rem.	4	7	11	11
Cond. Neg.	12	21	24	24
Total	32	46	58	59

Table 5. Faults found after using the invariant that keeps the current side active (results using the active side are indicated with an (a))

when defining the notion of decision coverage in Section 5.

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

As mentioned earlier, two test inputs $(b, \neg c, \neg d)$ and $(\neg b, \neg c, \neg d)$ would achieve decision coverage of the decision on statement 1, but there is no requirement that the outcome of the decision on statement 1 affects the flow of the program on statement 3. In fact, there is no requirements that the decision under consideration is even evaluated. Consider a minor modification of the example above where we break out the decision on line 1 to a Boolean function.

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

If the language has lazy-evaluation, `A()` will not get called and the decision (now on line 2) will not even get evaluated. Technically, however, the two test inputs $(b, \neg c, \neg d)$ and $(\neg b, \neg c, \neg d)$ would still achieve 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. This is exactly the problem we face with the condition based coverage criteria we have considered such as, 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 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. As can be seen in the data in Table 1, decision coverage 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 this problem, the coverage criteria must be modified to take data flow into account—the criteria must assure that somehow the decisions are invoked. The unanswered question is "How?". We must modify the requirements on the test suite to require that the decision is exercised in some way by the test suite. For example, the following are possibilities:

1. Decision invoked at least once for each required truth assignment,

2. Every invocation of the decision is tested for each required truth assignment,

3. Every invocation is invoked for each truth assignment of the decision, and the invocation point must be demonstrated to have an independent impact on the control flow.

As can be seen above, there are many combinations of condition based and data-flow based coverage criteria that can be developed. The question is, which ones are likely to reveal faults and which ones will give us test suites that are of a size that makes the test effort tractable? What we have done in this limited study is to demonstrate that the basic coverage criteria are clearly inadequate. If given he same amount of resources, random testing performs significantly better—there is a clear problems with the structural specification coverage criteria we evaluated that must be addressed.

8. Summary and Conclusions

To summarize, in a previous investigation we evaluated how well model checking techniques scaled when used for test case generation from formal specifications. [11]. 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. We are currently investigating these issues and hope to have additional results shortly. We do, however, recognize some of the limitations of our experiments. First, although we use a realistic model of the FGS as a case example, we have not conducted experiments on other models having a different structure. Second, we conducted the case study with quite simple condition based coverage criteria such as transition and decision coverage. We are encouraging other researchers to conduct similar experiments on specifications with different structure so the community can identify a collection of coverage criteria likely to reveal faults in a broad class of systems while at the same time being tractable to generate and execute.

References

[1] *Proceedings of The First International Workshop on Automated Program Analysis, Testing and Verificaiton, ICSE 2000*. 2000.

[2] P. E. Ammann and P. E. Black. A specification-based coverage metric to evaluate test sets. In *Proceedings of the Fourth IEEE International Symposium on High-Assurance Systems Engineering*. IEEE Computer Society, Nov. 1999.

[3] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 46–54. IEEE Computer Society, Nov. 1998.

[4] J. Callahan, F. Schneider, and S. Easterbrook. Specification-based testing using model checking. In *Proceedings of the SPIN Workshop*, August 1996.

[5] J. Chilenski and S. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9:193–200, September 1994.

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

[7] A. Engels, L. M. G. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In *Proceedings of TACAS'97, LNCS 1217*, pages 384–398. Springer, 1997.

[8] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. *Software Engineering Notes*, 24(6):146–162, November 1999.

[9] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[10] K. Hayhurst, D. Veerhusen, and L. Rierson. A practical tutorial on modified condition/decision coverage. Technical Report NASA/TM-2001-210876, NASA, 2001.

[11] M. P. Heimdahl, S. Rayadurgam, W. Visser, G. Devaraj, and J. Gao. Auto-generating test sequences using model checkers: A case study. In *3rd International Worshop on Formal Approaches to Testing of Software (FATES 2003)*.

[12] H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *Proceedings of 2003 International Confernece on Software Engineering*, Portland, Oregon, May 2003.

[13] H. S. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '02)*, Grenoble, France, April 2002.

[14] N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering*, 20(9):684–706, September 1994.

[15] NuSMV: A New Symbolic Model Checking. Available at http://nusmv.irst.itc.it/.

[16] S. Rayadurgam and M. P. 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.

[17] RTCA. *Software Considerations In Airborne Systems and Equipment Certification*. RTCA, 1992.

[18] F. C. A. S. Team. What is a "decision" in application of modified condition/decision coverage and decision coverage (dc)? Technical Report position paper, 2002.

[19] J. M. Thompson, M. P. Heimdahl, and S. 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.

[20] J. M. Thompson, M. W. Whalen, and M. P. Heimdahl. Requirements capture and evaluation in NIMBUS: The light-control case study. *Journal of Universal Computer Science*, 6(7):731–757, July 2000.