Test-Suite Reduction for Model Based Tests: Effects on Test Quality and Implications for Testing

Mats P.E. Heimdahl and Devaraj George

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

Abstract

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 MCDC. Test-suite reduction can provide us with a smaller set of test cases that preserve the original coverage—often a dramatically smaller set. One potential drawback with test-suite reduction is that this might affect the quality of the test-suite in terms of fault finding. Previous empirical studies provide conflicting evidence on this issue. To further investigate the problem and determine its effect when testing 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. Our results show that the size of the specification based test-suites can be dramatically reduced and that the fault detection of the reduced test-suites is adversely affected. In this report we describe our experiment, analyze the results, and discuss the implications for testing based on formal specifications.

Keywords: specification-based testing, test reduction, fault finding, model checkers, automated test generation

1 Introduction

In model-based development, the development effort is centered around a formal description of the proposed software system. The main ideas behind model-based development is that through manual inspections, formal verification, and simulation and testing we convince ourselves (and any regulatory agencies) that the software specification possesses desired properties. The implementation is then automatically generated from this specification and, in theory, little or no additional testing of the implementation is required.

With the use of formal models comes the ability to automatically generate specification based tests from the models. This capability may be used to generate large numbers of tests to use as *conformance tests* to provide assurance that the generated code is correct with respect to the specification from which it was generated. This type of conformance testing will most likely be required since it is unlikely that regulatory agencies will trust a complex code generation tool. For example, we may generate test-suites that provide MC/DC coverage of the formal model, execute the tests on the generated code, and show that the specification and code behave equivalently for this test-suite—an argument for the correctness of the translation that may be accepted by a regulatory agency.

The cost of generating, executing, storing, and maintaining these test-suites can be reduced through *test-suite reduction techniques*. Test-suite reduction aims to remove (or not generate at all) test-cases from a test-suite in such a way that "redundant" test-cases are eliminated. For example, a reduced test-suite T_R may provide the same structural coverage as a test-suite T with significantly fewer test-cases. Previous studies conducted on C code have shown that test-suite reduction techniques significantly reduce the number of test-cases in a test-suite while maintaining the structural coverage of the original suite [34, 35, 30, 24]. The effect on the *fault finding* capability of the reduced test-suites is, however, unclear and the studies show conflicting evidence. Wong *et al.* [34, 35] found no significant effect in fault finding ability between the full suites and the reduced suites. On the other hand, Rothermel *et al.* [30] and Jones and Harrold [24] showed that the reduced test-suites can be dramatically worse with respect to fault finding.

To investigate the effect of test-suite reduction in the domain of automatically generated conformance test-suites, we conducted an experiment where we compared the test-suite size and fault finding capability of reduced test-suites generated to six different specification test-adequacy criteria. As a system-under-test, we used a model of a production sized Flight Guidance System (FGS) provided by Rockwell Collins Inc. in which we seeded "representative" faults; faults we had observed during the development of the FGS model.

Our results show that one can dramatically reduce our automatically generated conformance test-suites while maintaining desired coverage. We also found that the fault finding of these reduced test-suites was adversely affected, and that the reduction is quite significant in the domain of specification based testing. Although further studies are needed, the results indicate that test-suite reduction may not be an effective means of reducing testing effort—the cost in terms of loss in fault finding capability is too high.

In the remainder of the paper we review relevant literature, describe our experimental set up, results obtained, and draw conclusions from the results.

2 Background and Related Work

To put our current work in context it is necessary to provide information regarding related studies as well as the domain in which we performed our work. We will briefly discuss the approach to testing made possible when working with formal models and automatic test case generators. We will then cover the most closely related test-suite reduction experiments and contrast them with the study presented in this report.

2.1 Model-Based Development

As mentioned in the introduction, in the embedded systems community, there is a trend towards *model-based* [3, 33] (or specification based) development. In model-based development, the development effort is centered around a formal description of the proposed software system. For validation and verification purposes, this *formal specification* can then be subjected to various types of analysis, for example, completeness and consistency analysis [19, 21] model checking [14, 6, 7, 22, 9], theorem proving [1, 2], and test case generation [5, 13, 10, 4, 27, 23, 29]. Through manual inspections, formal verification, and simulation and testing we convince ourselves (and any regulatory agencies) that the software specification possesses desired properties. The implementation is then automatically generated from this specification. There are currently several commercial and research tools that attempt to provide these capabilities—commercial tools are, for example, Esterel and SCADE from Esterel Technologies, Statemate from i-Logix [15], and SpecTRM from Safeware Engineering [26]; and examples of research tool are SCR [20], RSML^{-e} [33], and Ptolemy [25].

The capabilities of model-based development allows us to follow a process outlined in Figure 1. The testing effort has in this process been largely moved from unit testing of

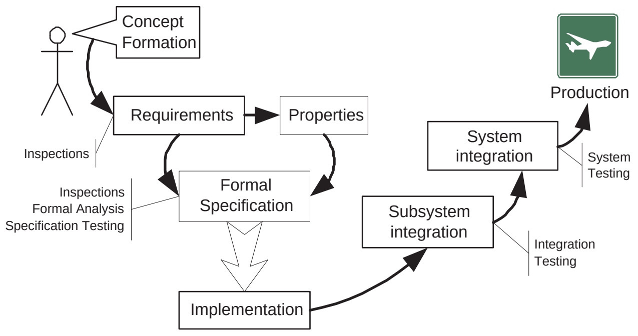

Figure 1. Specification Centered Development Process.

the code to functional testing of the formal model. In addition, there is a need to perform conformance testing to assure that the generated code is behaviorally equivalent to the specification—a task ideally suited for automatic test case generation from the formal specification. Test-suites generated for this purpose are the focus of our study.

2.2 Test Cases and Model Checkers

Model checkers build a finite state transition system and exhaustively explore the reachable state space searching for violations of the properties under investigation [9]. 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* [13]. 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. This general approach can be used to generate tests for a wide variety of structural coverage criteria, such as all state variables have taken on every value, and all decisions in the model have evaluated to both true and false, etc.

In a previous project, we developed a framework where one can generate test suites to satisfy a wide variety of specification-coverage criteria [29, 18]. This is the technique and tools infrastructure we have used to generate the test suites used in our experiment.

2.3 Previous Test-Reduction Experiments

Several studies have investigated the effect of test-set reduction on the size and fault finding capability of a test-set. In an early 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 [34, 35]. 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.

To confirm or refute the results in the Wong study, Rothermel *et al.* performed a similar experiment using seven sets of C programs with manually seeded faults [30]. For their experiment they used edge-coverage [11] 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 [24].

These radically different results are difficult to reconcile and the relationship between coverage criteria, test-suite size, and fault finding capability clearly needs more study.

In the experiment discussed in this paper we attempt to shed some additional light on this issue. Our work is different in some respects, however. First, we are not studying testing of traditional programs, we are interested in test-case generation and testing of formal specifications. In particular, formal specifications expressed in synchronous data-flow languages commonly used in model-based development, for example, Esterel, SCADE, SpecTRM, SCR, and RSML^{-e}.

Second, we are addressing a wide spectrum of coverage criteria ranging from the very weak, for example, transition coverage, to the very strong, for example MCDC. The previous experiments addressed either rather weak criteria such as block-coverage [35] or used test-suites that did not fully provide the desired strong coverage [24]. This issue will be further addressed in the discussion of our results.

These differences makes a direct comparison of our results with related work difficult, but our findings seem to reinforce the observations in the Rothermel *et al.*, and Jones and Harrold studies; although test-suite reduction can dramatically reduce the size of a test-suite without affecting coverage, test-suite reduction has a detrimental effect on the test-suite's fault finding capability.

3 The Experiment

To investigate the relationship between test reduction and fault finding capability in the domain of model based tests, we designed our experiment to test 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 [18], we believe this generalizes to other criteria as well. (A discussion of the various specification coverage criteria will follow in Section 3.4 below.) 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.

3.1 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 reveals faults as compared to a reduced test-suite providing the same coverage. To provide realistic results, we conducted the experiment using a close to production model of a flight guidance system from Rockwell Collins Inc.[1]

[1]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.

We conducted the experiment through the the steps outlined below. Each step is elaborated in detail in the following sections.

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 straight forward to generate, but they were also highly redundant.
2. We generated 100 faulty specifications of the FGS by randomly seeding one fault per faulty specification. The fault classes we seeded are discussed in Section 3.3.
3. We ran the full (non-reduced) test suite on the 100 faulty specifications and recorded the number of faults revealed.
4. We generated and ran five reduced test suites for each full test-suite, ensuring that the desired coverage criterion was maintained. As discussed below, we generated five reduced sets for each full test-suite to avoid skewing our results because we were lucky (or unlucky) in the selection of tests for a reduced test-suite.
5. Given the results of the previous steps, we compared the relative fault finding capability of the full test-suites versus the reduced test-suites.

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

3.2 Case Example: The FGS

A Flight Guidance System (FGS) is a component of the overall Flight Control System (FCS) in a commercial aircraft. The FGS was developed using the RSML^{-e}language. 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 2 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

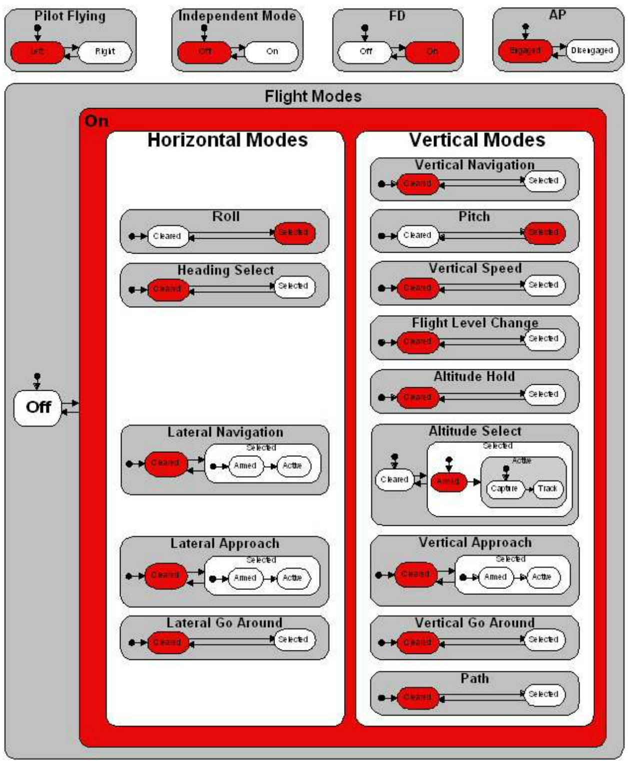

Figure 2. Flight Guidance System

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, but does not represent any actual Rockwell Collins product. The model consists of 2564 lines of code in RSML^{-e}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.

3.3 Fault Injection and Detection

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

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

Condition Insertion (CI): 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 (CR): A condition that was previously required to be true (T) or false (F) in a table was changed to "don't care" (*).

Condition Negation (CN): 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).

During our testing experiment, we used an 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. Thus, our oracle was able to detect faults that may not have manifested themselves as erroneous outputs, but only as a corrupt model state. We chose this approach since we expect this to be the type of oracle used when performing conformance testing of auto-generated code.

3.4 Specification Based Test Criteria

Adequacy criteria are used by testers to decide when to stop testing by helping them determine if the software has been adequately tested. In this paper, these criteria are defined on a synchronous data-flow specification language. We are using the specification language RSML^{-e} [33]in our study, 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 Σ 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. In this paper we use the following six specification coverage criteria. 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.

Variable Domain Coverage: (Often referred to as state-coverage.) Requires that the test set Γ has test-cases that enable each control variable defined in the model Σ 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 Γ has test-cases that exercise every transition definition in Σ at least once.

Decision Coverage: Each decision occurring in Σ evaluates to true at some point in some test-case and evaluates to false at some point in some other test case. Note that if the decision is, for example, in a function, there is no requirement that the function is actually invoked—this criterion only requires that the decision would have evaluated to true/false if it was evaluated during the test case.

Decision Coverage with Single Uses: Analogous to decision coverage, but the decision must actually be evaluated. For example, for a condition in a function, the condition must evaluate to true/false while the function is invoked from some point in the model.

Modified Condition and Decision Coverage (MCDC): Every condition within the decision has taken on all possible outcomes at least once, and every condition has been shown to independently affect the decision's outcome. Note again that invocation of the decision is not required.

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

The reader may wonder why we have included the coverage criteria that do not require that decisions are actually evaluated. These criteria are included because there is no consensus if the definitions of the coverage criteria in, for example, DO-178B [31], require the decisions to be evaluated or not. A more formal treatment of these coverage criteria can be found in in [28, 29] and [16, 32].

3.5 Test Set Generation and Reduction

We generated full test-suites using the approach discussed in Section 2.2. We used the NIMBUS tool-set (an execution and analysis environment for RSML^{-e}) to translate to the input language of NuSMV [8] 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 in most cases may satisfy 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

```
Algorithm 3.1: TEST-REDUCE($\Sigma, \Gamma, \eta$)

 INPUTS :
   Model $\mathbf{\Sigma}$, test suite $\mathbf{\Gamma}$, and test criterion $\eta$
 OUTPUT :
   Reduced test set $\mathbf{\Omega}$

 $\Omega \leftarrow \emptyset$;   ReducedTest set
 $AC \leftarrow 0$;   Actual Coverage
 $PC \leftarrow 0$;   Previous Coverage
 $shuffle(\Gamma)$;
 repeat
   choose a test case $f$ from $\Gamma$;
   run $f$ against the model $\Sigma$;
   Measure actual coverage $AC$;
   if $AC \neq PC$
     then $\Omega \leftarrow \Omega \cup \{f\}$;
   $PC \leftarrow AC$;
 until $\Gamma$ is exhausted
 return $(\Omega)$;
```

Figure 3. Algorithm for test-suite reduction.

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 a NP problem [12], 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 3). 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.

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

4 Experimental Results and Analysis

As a baseline for our experiments, we ran the full test-suites as well as a randomly generated sets-set. The results are summarized in Table 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).

There are several things worth noting about Table 1. First, we did not attempt to eliminate specifications where the seeded fault did not yield a behaviorally different specification. Thus, the numbers do not say anything about the *absolute fault finding capability* of the various coverage criteria; we can only evaluate the *relative fault finding capability*. Nevertheless, to get a basic idea of the fault finding capability of the test-suites designed to provide the various structural coverage, we also created a collection of randomly generated tests. We expended approximately 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. As can be seen in Table 1, the randomly generated test perform surprisingly well compared to the test-suites providing structural coverage. We have discussed the reasons behind the poor performance of Variable Domain and Transition Coverage in a previous study [17] and a discussion of this topic is outside the scope of this paper.

From the results in Table 1 one can also observe that the more rigorous the test criteria, the better the fault finding capability. For instance, MCDC with usage detects more faults (72%) than any other coverage criteria considered and also outperforms random testing (66%).

4.1 Test-Suite Reduction

As mentioned earlier, we generated five different reduced test-suites to control the possibility that we by chance got a very "good" or very "poor" reduced test-suite. The results of the reduction algorithm can be seen in Table 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 [34, 35, 30, 24] and is to be expected since our test-case generation method produces a significant number of overlapping test-cases; we generate a separate test-case for each construct of interest. Of more interest is the fault finding ability of the reduced tests-suites discussed next.

4.2 Effect on Fault Detection Effectiveness

The fault finding capability of the full as well as reduced test-suites is summarized in Table 3. The results are in agreement with our second hypothesis that test-suit reduction will adversely impact the fault finding ability of test-suites that are derived from synchronous data-flow models.

Test Criteria	Size	VR	CN	CI	CR	Total
Random	100	21	25	5	15	66
Variable Domain	115	14	15	2	4	32
Transition	313	20	24	5	15	64
Decision	435	23	24	5	15	67
Decision Usage	478	23	24	7	15	69
MCDC	537	22	25	7	16	70
MCDC Usage	334	23	25	8	16	72

Table 1. Full test set generation for various criteria along with their fault detection capability

Criteria	Full Set	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Reduction
Variable Domain	115	19	22	18	21	21	20.2	82%
Transition	313	35	43	29	38	43	37.6	88%
Decision	435	45	44	44	45	42	44.0	90%
Decision Usage	478	37	43	47	43	38	41.6	91%
MCDC	537	34	33	29	34	32	32.4	94%
MCDC Usage	334	30	30	33	32	33	31.6	91%

Table 2. Reduced test set sizes for various test reduction runs

As shown in Table 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.

From our results we can also observe that the most rigorous coverage criteria, MC/DC with Usage, seems to be the least sensitive to the effect of test-suite reduction. We speculate that this is because it is simply harder to come up with a test-suite that provides this high level of coverage without finding faults–MC/DC with Usage is simply a "better" coverage criterion than the other ones we used in our experiment. We hypothesize that MC/DC with Usage is better than the other criteria in two respects. First, it seems to find more faults than any other criteria. Second, it seems to be less sensitive to the effect of test-suite reduction. 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 [35, 24, 30]; one of the studies reports no reduction in fault finding and two studies report a dramatic and 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% [24, 30].

We do not have a ready explanation for this phenomenon, but we speculate that it may be related to two factors; (1) the coverage criteria used in the experiment and (2) the actual coverage provided by the test-suites. The Rothermel *et al.* study [30] used edge-coverage of the control flow graph (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 [24] 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 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. 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 capa-

Criteria	Full Set	Run 1	Run 2	Run 3	Run 4	Run 5	Average	Reduction
Variable Domain	32	28	29	25	28	25	27.0	15.6%
Transition	64	58	58	58	59	57	58.0	9.38%
Decision	67	62	61	62	62	61	61.6	8.06%
Decision Usage	69	62	63	63	62	63	62.6	9.28%
MCDC	70	64	63	63	63	63	63.2	9.71%
MCDC Usage	72	67	66	67	67	67	66.8	7.22%

Table 3. Fault finding capability of the reduced test-sets

bility. 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* [24] 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.

4.3 Threats to Validity

There are three 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 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 and our results raise serious doubts about the use of any test-suite reduction techniques in this domain.

5. Summary and Conclusions

We have described an 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 automatically generated conformance test-suites while maintaining desired coverage, and the fault finding of the reduced test-suites was adversely affected. Although we cannot broadly generalize our results and further studies are needed, the experiment indicates that test-suite reduction 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.

Furthermore, 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.

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 loss in fault finding capability.

References

[1] M. Archer, C. Heitmeyer, and S. Sims. TAME: A PVS interface to simplify proofs for automata models. In *User Interfaces for Theorem Provers*, 1998.

[2] S. Bensalem, P. Caspi, C. Parent-Vigouroux, and C. Dumas. A methodology for proving control systems with Lustre and

PVS. In *Proceedings of the Seventh Working Conference on Dependable Computing for Critical Applications (DCCA 7)*, 1999.

[3] V. Berzins, Luqi, and A. Yehudai. Using transformations in specification-based prototyping. *IEEE Transactions on Software Engineering*, 19(5):436–452, May 1993.

[4] M. R. Blackburn, R. D. Busser, and J. S. Fontaine. Automatic generation of test vectors for SCR-style specifications. In *Proceedings of the 12th Annual Conference on Computer Assurance, COMPASS'97*, June 1997.

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

[6] W. Chan, R. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.

[7] Y. Choi and M. Heimdahl. Model checking RSML^{-e} requirements. In *Proceedings of the 7th IEEE/IEICE International Symposium on High Assurance Systems Engineering*, October 2002.

[8] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A New Symbolic Model Checker, 2004. Available at http://nusmv.irst.itc.it/.

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

[10] 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.

[11] P. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. In *Proceedings of the symposium on Testing, analysis, and verification*, 1991.

[12] M. Garey and D. Johnson. *Computers and Intractability*. Freeman, New York, 1979.

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

[14] O. Grumberg and D.E.Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[15] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

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

[17] M. P. Heimdahl, G. Devaraj, and R. J. Weber. Specification test coverage adequacy criteria = specification test generation inadequacy criteria? In *Proceedings of the Eighth IEEE International Symposium on High Assurance Systems Engineering (HASE)*, Tampa, Florida, March 2004.

[18] 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)*, 2003.

[19] M. P. E. Heimdahl and N. G. Leveson. Completeness and consistency in hierarchical state-base requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.

[20] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proceedings of the Tenth Annual Conference on Computer Assurance, COMPASS 95*, 1995.

[21] C. Heitmeyer, R. Jeffords, and B. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

[22] C. Heitmeyer, J. K. Jr., B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Transactions on Software Engineering*, 24(11):927–948, November 1998.

[23] R. Jasper, M. Brennan, K. Williamson, B. Currier, and D. Zimmerman. Test data generation and feasible path analysis. In *Proc. of Int'l Symp. on Software Testing and Analysis*, pages 95–107, August 1994.

[24] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Emgineering*, 29(3):195–209, March 2003.

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

[26] N. G. Leveson, M. P. Heimdahl, and J. D. Reese. Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the Future. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, volume 1687 of *LNCS*, pages 127–145, September 1999.

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

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

[29] 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.

[30] G. Rothermel, M. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. *Proceedings of the International Conference on Software Maintenance*, pages 34–43, November 1998.

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

[32] 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.

[33] 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 Soft-*

ware Engineering, number 1687 in LNCS, pages 163–179, September 1999.

[34] W. Wong, J. Horgan, A. Mathur, and A.Pasquini. Test set size minimization and fault detection effectiveness: A case study in a space application. *Proceedings of the 21st Annual International Computer Software and Applications Conference*, pages 522–528, August 1997.

[35] W. Wong, J. Horgan, S.London, and A. Mathur. Effect of test set minimization on fault detection effectiveness. *Software Practice and Experience*, 28(4):347–369, April 1998.