ON MC/DC AND IMPLEMENTATION STRUCTURE: AN EMPIRICAL STUDY

Mats P.E. Heimdahl, University of Minnesota, Minneapolis, Minnesota
Michael W. Whalen, Rockwell Collins Inc., Cedar Rapids, Iowa
Ajitha Rajan, Matt Staats, University of Minnesota, Minneapolis, Minnesota

Abstract

In civil avionics, obtaining D0-178B certification for highly critical airborne software requires that the adequacy of the code testing effort be measured using a structural coverage criterion known as Modified Condition and Decision Coverage (MC/DC). We hypothesized that the effectiveness of the MC/DC metric is highly sensitive to the structure of the implementation and can therefore be problematic as a test adequacy criterion. We tested this hypothesis by evaluating the fault-finding ability of MC/DC-adequate test suites on five industrial systems (flight guidance and display management). For each system, we created two versions of the implementations—implementations with and without expression folding (i.e., inlining).

We found that for all five examples, the effectiveness of the test suites was highly sensitive to the structure of the implementation they were designed to cover. MC/DC test suites adequate on an inlined implementation have greater fault finding ability than test suites generated to be MC/DC adequate on the non-inlined version of the same implementation at the 5% significance level. (The inlined test suites outperformed the non-inlined test suites in the range of 10% to 5940%.)

This observation confirms our suspicion that MC/DC used as a test adequacy metric is highly sensitive to structural changes in the implementation, and that test suite adequacy measurement using the MC/DC metric will be better served if done over the inlined implementation.

Introduction

Test adequacy metrics defined over the structure of a program, such as statement coverage, branch coverage, and decision coverage, have been used for decades to assess the adequacy of test suites. Nevertheless, it is well known that these criteria can easily be ‘cheated’ by restructuring a program to make it easier to achieve the desired coverage. For several years, we have been particularly interested in the Modified Condition and Decision Coverage (MC/DC) criterion [1] since it is used when testing highly critical software in the avionics industry [2]. Of particular interest is the possible use of MC/DC in conjunction with model-based development using tools such as Simulink [3] and SCADE [4]. Such tool adoption has led to a discussion on what coverage criteria to use when testing such models; MC/DC has been a natural candidate for adoption for the most critical models.

Our concern regarding MC/DC is its sensitivity to the structure of the program or model under test. A test suite provides MC/DC over the structure of a program or model if every condition within a decision has taken on all possible values at least once, and every condition has been shown to independently affect the decision’s outcome (note here that when discussing MC/DC coverage, a decision is defined to be an expression involving any Boolean operator). Consider the trivial program fragments in Table 1[1]. The program fragments have different structures but are functionally equivalent. Version 1 is non-inlined with intermediate variable `expr_1`, Version 2 is inlined with no intermediate variables. Based on the definition of MC/DC, `TestSet1` in Table 1 provides MC/DC over program Version 1 but not over Version 2; the test cases with `in_3 = false` (bold faced) contribute towards MC/DC of `in_1 or in_2` in Version 1 but not over Version 2 since the masking effect of `in_3 = false` is revealed in Version 2.

[1] Note that the example discussed in the next few paragraphs has been adopted from our pilot study [5].

In contrast, MC/DC over the inlined version requires a test suite to take the masking effect of `in_3` into consideration as seen in `TestSet2`. This disparity in MC/DC coverage over the two versions can have significant ramifications with respect to fault finding of test-suites. Suppose the code fragment in Table 1 is faulty and should have been `in_1 and in_2` (which was erroneously coded as `in_1 or in_2`). `TestSet1` would be incapable of revealing this fault, since there would be no change in the observable output—`out_1`. On the other hand, any test set providing MC/DC of the inlined implementation would be able to reveal this fault.

Version 1: Non-Inlined Implementation

```
expr_1 = in_1 or in_2;         //stmt1
out_1 = expr_1 and in_3;       //stmt2
```

Version 2: Inlined Implementation

```
out_1 = (in_1 or in_2) and in_3;
```

Sample Test Sets for (in_1, in_2, in_3):

```
TestSet1 = {(TFF),(FTF),(FFT),(TTT)}
TestSet2 = {(TFT),(FTT),(FFT),(TFF)}
```

Table 1: Example of behaviorally equivalent implementations with different structures.

Programs may be structured with significant numbers of intermediate variables for many reasons, for example, for clarity (nice program structure), efficiency (no need to recompute commonly used values), or to make it easer to achieve the desired MC/DC coverage (it is significantly easier to find the MC/DC tests if the decisions are simple). Either way, we hypothesize the efficacy of the MC/DC coverage criterion will be reduced over such programs.

In a previous study, we examined the effect of program structure by comparing test suites necessary to cover programs consisting of simple decisions (at most one logical operator) versus programs in which such intermediate variables are removed [5]. We refer to these versions as the non-inlined and inlined implementations, respectively. We found the effect of such transformations to be dramatic: our analysis revealed that the transformations yield an average reduction of 29% in MC/DC coverage measured over inlined implementations (which was statistically significant for a null hypothesis of no difference at the 5% significance level). Given that the inlined and non-inlined implementations are semantically equivalent, the discrepancy in coverage under this simple transformation was a cause for serious concern.

Of course, the real concern is not the reduction in coverage itself, but whether this reduction corresponds to a drop in the fault finding capability of the test suite. In this paper, we conduct an experiment to investigate whether or not the structure of the implementation used for test case generation to MC/DC has any effect on the fault finding potential of the test suites.

For our fault finding experiment we used five industrial systems from the civil avionics domain. These systems were modeled using the Simulink language. From the models we created implementations that we used as a basis for the generation of large sets of mutants with randomly seeded faults. We generated numerous test suites providing 100% achievable MC/DC coverage over *non-inlined* versions of the models (we call them *non-inlined test suites*) as well as test suites providing coverage of *inlined* versions of the models (*inlined test suites*). We assessed the fault finding capability of the non-inlined and inlined test suites by running them over the sets of mutants and measuring how many faults were found. We generated enough test suites and mutants to achieve statistical significance at the 5% significance level.

Our experiment revealed that the structure of the implementation used for test case generation has a profound effect on the fault finding of the resultant test suites; the inlined test suites performed significantly better for all five examples in our case study. The relative improvement in fault finding from non-inlined to inlined test suites ranged from 10% to 5940% on these examples. Statistical analysis of these results revealed that for all systems, our hypothesis that the MC/DC metric is sensitive to the structure of the implementation is supported.

Note here that the potential problems with decision structure are not confined only to code; the move towards model-based development in the avionics community makes test-adequacy measurement a crucial issue in the modeling domain. Coverage criteria, in particular MC/DC, are being used in conjunction with modeling tools such as Simulink [3] and SCADE [4] for testing models. Currently, MC/DC measurement over models in these tools is being done in the weakest possible manner. For example, Figure 1 is a Simulink model equivalent to the example in Table 1. MC/DC coverage of such models is currently defined on a 'gate level' (analogous to the MC/DC measurement over Version 1 in Table 1). Since there are no complex decisions in this definition of MC/DC, MC/DC measured this way is susceptible to the masking problem discussed above, and test-suites designed to provide MC/DC coverage over the models may therefore provide poor fault-finding capability. Thus, the current approach to measuring MC/DC over such models is a serious cause for concern. For simplicity, in the remainder of this paper, we refer to a 'model' or 'program' as the 'implementation' since the concerns discussed here are the same regardless of whether we are discussing a model or a program.

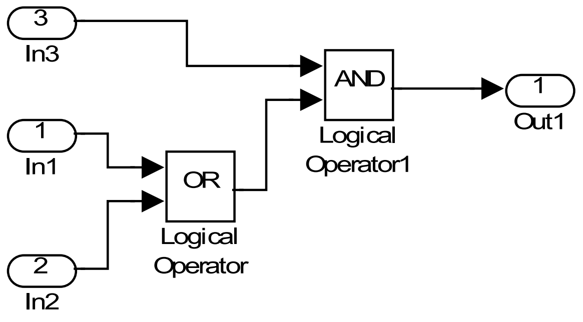

Figure 1: Simulink model of example in Table 1.

We find the effect of program or model structure on the fault finding of test suites providing MC/DC disconcerting. In particular, we are concerned about the efficacy of test suites automatically generated from models where MC/DC is measured at a 'gate level'. Engineers and certifiers must be aware and cautious of this issue when using MC/DC to assess the adequacy of test suites for safety-critical applications; our results show that MC/DC applied to implementations with inappropriate structure can be ineffective.

Additionally, we also observed that the oracle used to assess test suite effectiveness can make a significant difference in the evaluation. We used two different oracles in our experiment; one that compares only the outputs of the mutated models to that of a reference model (output only oracle), and a second that compares both the outputs and internal state of the mutated and correct models (internal variable oracle). We found that the internal variable oracle revealed significantly more faults than the output only for all case examples. Thus, test suites are more effective when a stronger oracle such as the internal variable one is used. The oracle selection problem and its implication in fault finding merits more attention and we hope to take a closer look at this issue in our future work.

In the remainder of the paper we first introduce our experimental setup and case examples. We then in detail discuss the techniques used to generate the faulty implementations, the test oracles, and the test suites. Finally we provide a detailed discussion of our results and cover the implications of our results, and point to future directions.

Experiment

To investigate the effect of program structure on MC/DC coverage, we designed our experiment to test the following hypothesis:

Hypothesis: *A test-suite generated to provide MC/DC over an inlined implementation will reveal more faults than a test-suite generated to provide MC/DC over a non-inlined implementation.*

As mentioned in the introduction, we previously found that the structure of an implementation can have a significant impact on the MC/DC coverage achieved by a test suite; a test suite providing high coverage of a non-inlined implementation may provide significantly lower coverage of the same implementation if it were inlined. Based on this result, we believe that the structure of the implementation used as the basis for adequacy measures or test suite generation (manual or automated) can significantly impact the quality of the test suites.

Experimental Setup Overview

The goal in our experiments was to determine whether inlining the implementation had a significant effect in the fault finding capability of test suites generated to provide MC/DC over the implementation. To perform this assessment we first generate a test-suite to provide MC/DC over a non-inlined version of the implementation (referred to as *non-inlined test suite* from hereon); we also generate a test-suite to provide MC/DC over a inlined version of the same implementation (referred to as *inlined test suite* from hereon). We then reduce the inlined and non-inlined suites using a greedy algorithm to smaller test suites that maintain 100% achievable MC/DC coverage of the respective implementations. We then seed faults to create mutants (a single fault per mutant) and run the reduced inlined and non-inlined test suites against the correct and mutated implementations to compute their fault finding capability (defined to be the number of mutants caught over the total number of mutants seeded). It is now possible to determine whether a program in which complex decisions have been split into simple decisions reduces the fault finding effectiveness of the MC/DC coverage criterion. Below we provide an overview of the experimental setup, subsequent sections provide detailed information on the experimental procedure.

We conducted experiments on five industrial examples: three models from a display window manager for an air-transport class aircraft DWM_1, DWM_2, and DWM_3), and two models representing flight guidance mode logic for a business and regional jets class aircraft (Vertmax_Batch and Latctl_Batch). (A description of the case examples follow in the next section). For each of the case examples, given the inlined and non-inlined test suites we varied the following parameters:

Mutant Set: We randomly generated three sets of 200 mutants for each model.

Test Suites: We used test suites of two types: 'inlined' and 'non-inlined.' For each type, we randomly created three reduced test suites that maintain MC/DC over the respective implementation (non-inlined implementation for non-inlined test suite and inlined implementation for inlined test suite.)

Oracle: For reasons that will be described in detail shortly, we conducted our experiment using two different oracles. One oracle only considers values of 'system level' outputs in the comparison between a mutant and a correct implementation, and one oracle considers both values of internal variables (also referred to as intermediate variables) and outputs in the comparison.

Thus, for each case example, we had 3 sets of mutants, 3 randomly reduced inlined test suites, 3 randomly reduced non-inlined test suites and two oracles. For each of the two oracles, we have 9 observations each for the inlined and non-inlined test suites. We designed our experiment using multiple sets of mutants and multiple sets of test suites to reduce the influence of outliers on our results.

We conducted the experiment using the following steps.

Step 1: From the specification of a case example, we generated two artifacts:

An implementation with no inlining. The structure of this implementation closely reflects the structure of a typical Simulink model (very simple conditions and many intermediate variables carrying temporary values). We will refer to this version of the implementation as the 'non-inlined' version.

An implementation that is inlined. The implementation is inlined in the sense that multiple levels of hierarchy in the specification are flattened to a single level (functions calls are inlined) and intermediate variables in complex conditions are inlined. This level of inlining is similar to the standard options for the Simulink RTW [3] system, and is described in more detail below. We will refer to the implementation generated in this manner as the 'inlined' version. In our experiment we use the 'inlined' version of the implementation as the 'oracle implementation' that is used in the test runs to compare against mutants.

Step 2: We generated test suites from both the non-inlined and inlined implementations to provide maximal MC/DC coverage on each. The test suites were naively generated (one test case per MC/DC test obligation) which led to much larger test suites than needed to provide maximal coverage.

Step 3: For each of the test suites in Step 2, we generated three randomly reduced test suites that maintained MC/DC coverage. We used a simple greedy approach for the reduction.

Step 4: We randomly generated three sets of 200 mutants from the oracle implementation using the method outlined in the next section.

Step 5: We ran each of the reduced test suites for both inlined and non-inlined against each set of mutants using both test oracles, and recorded the percentage of mutants caught.

Case Examples

In our experiment, we used five close to production or production. All systems used in our experiment were modeled using the Simulink notation from Mathworks Inc.

Flight Guidance System

A Flight Guidance System is a component of the overall Flight Control System (FCS) in a commercial aircraft. It compares the measured state of an aircraft (position, speed, and altitude) to the desired state and generates pitch and roll-guidance commands to minimize the difference between the measured and desired state. The FGS consists of the mode logic, which determines which lateral and vertical modes of operation are active and armed at any given time, and the flight control laws that accept information about the aircraft's current and desired state and compute the pitch and roll guidance commands. The two FGS models in this paper focus on the mode logic of the FGS. The Vertmax_Batch and Latctl_Batch models describe the vertical and lateral mode logic for the Rockwell Collins FCS 5000 flight guidance system family.

Display Window Manager Models (DWM_1, DWM_2 and DWM_3)

The Display Window Manager models, DWM_1, DWM_2 and DWM_3, represent 3 of the 5 major subsystems of the Display Window Manager (DWM) of the Rockwell Collins ADGS-2100, an air transport-level commercial displays system. The DWM acts as a 'switchboard' for the system and has several responsibilities related to routing information to the displays and managing the location of two cursors that can be used to control applications by the pilot and copilot. The DWM must update which applications are being displayed in response to user selections of display applications, and must handle reversion in case of hardware or application failures, deciding which information is most critical and moving this information to the remaining display(s).

Inlined and Non-Inlined Implementations

The case examples were modeled in Simulink and we will refer to these models as the *specification* of the system. As part of a previous project, we developed a translation framework with the ability to translate Simulink Models into the synchronous programming language Lustre [6]. Lustre is a synchronous dataflow language and is the underlying notation for the SCADE Suite from Esterel Technologies [4]. We translated each of the case examples modeled in Simulink to Lustre. We will refer to the translated case examples in Lustre as the *implementation* of the specification. This is analogous to automated code generation (with default compilation options) from Simulink models using Real Time Workshop from Mathworks [3], where the generated C code is the implementation of the Simulink specification. Using the options in our translation infrastructure, we generated two different implementations in Lustre—with and without inlining, as described below.

No Inlining

The structure of the generated non-inlined implementation in Lustre closely follows the structure of the specification in Simulink in terms of the hierarchies (or subsystems) and intermediate variables needed to propagate signals in the Simulink model. A sample implementation with no inlining is presented in Table 2. Note that the example is presented in a C-like language rather than Lustre for clarity; the structure (but not syntax) of the corresponding Lustre code would look similar. The example implementation computes the result of `X and Y or Z` for the three inputs `X`, `Y`, `Z` if the danger condition (`temp > thresh`) is not violated.

With Inlining

Here we flatten the multiple levels of hierarchy in the Simulink model of the system so that the implementation in Lustre has only a single level of hierarchy. In addition we also inline intermediate variables in the model into their original definition. Note however that we do not inline all the intermediate variable definitions. MC/DC was defined for constructs in a traditional imperative language such as C. We therefore made an attempt to make inlining in Lustre resemble that of an imperative language. For instance, although 'if-then-else' constructs are expressions in Lustre and can thus be inlined, such inlining would not be possible in C where 'if-then-else' is a statement. Thus, we did not inline variables defined through an 'if-then-else' expression. Table 2 presents the inlined version of the previously mentioned example.

Non-Inlined Implementation:

```
bool Compute(bool x, bool y, bool z,
             int temp, int thresh)
{
  bool run, no_danger, no_alarm;
  run = AndOr(x,y,z);
  no_danger = (temp<=thresh);
  if (no_danger) then {no_alarm=true;}
  else {no_alarm=false;}
  return (run && no_alarm);
}

bool AndOr(bool a, bool b, bool c)
{
  bool local;
  local = b || c;
  return (a && local);
}
```

Inlined Implementation:

```
bool Compute(bool x, bool y, bool z,
             int temp, int thresh)
{
  bool no_alarm;
  if (temp<=thresh) then {no_alarm=true;}
  else { no_alarm = false; }
  return ((x && (y || z)) && no_alarm);
}
```

Table 2: Example implementation with and without inlining.

Note that the terms '*Inlined Implementation*' and '*Non-Inlined Implementation*' used in the rest of this paper refers to the implementation in Lustre with and without inlining (as described above) respectively.

Mutant Generation

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

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

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

Boolean: Changes a boolean operator (Or, And, XOR).

Negation: Introduces the boolean NOT operator.

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

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

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

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

In our experiment, we generated mutants so that the 'fault ratio' for each fault class is uniform. The term fault ratio refers to the number of mutants generated for a specific fault class versus the total number of mutants possible for that fault class. For example, assume an implementation consists of 30 Relational operators and 275 Boolean operators. Thus there are 30 possible Relational faults and 275 possible Boolean faults. If we wish to generate 200 mutants from this implementation, our fault seeding tool using the uniform fault ratio distribution assumption would generate 66% each of the possible Relational and Boolean faults, which in this case translates to 19 Relational and 181 Boolean faults.

We generated three sets of 200 mutants for each case example. We generated three mutant sets for each example to reduce potential bias in our results from a mutant set that may have very hard (or easy) faults to detect.

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

Our mutant generator does not guarantee that a mutant will be semantically different from the original implementation. Nevertheless, this weakness in mutant generation does not affect our results, since we are investigating the relative fault finding of test suites, not the absolute fault finding.

Mutation testing using fault seeding approaches similar to the one described above have been shown to be an effective metric of a test suite's fault finding ability when realistic mutation operators are used [7].

Test Oracles

Given a test case, we need to determine if the system executed correctly for the test case. To do this, we use a test oracle that gives the expected outcome of the test. For this experiment, we choose two different oracles. The first compares only the outputs of the correct and mutated implementations. The second compares both the outputs and the internal state. In general, internal state information of the system under test may not be available and it is therefore preferable to perform the comparison with only output values. Initially in our experiment, we only set out to compare 'outputs' between the mutants and the correct reference implementation. Nevertheless, such an oracle is incapable of revealing seeded faults that result in a corrupted state but does not propagate to outputs (because of masking). Therefore, we decided to additionally use a stronger oracle that compares internal state information in addition to outputs. To assess the impact of oracle selection on fault finding, we use both oracles (only outputs as well as outputs + intermediate variables) in our assessment of inlined versus non-inlined test suites on all case examples. Note that similar oracle issues have been explored previously [8], but not in the context of this domain.

When the oracle comparison is done using only outputs, we term it as an 'output oracle', and when we compare both internal state and outputs, we term it as an 'intermediate variable oracle' or 'IV oracle'.

Test Suite Generation and Reduction

The full inlined and non-inlined test suites used in the experiments were the same ones used in previous work [5]. The test suites were automatically generated using the NuSMV model checker [9] to provide MC/DC over the implementation. These full test suites were generated in a naive manner, with a separate test case for every construct we need to cover in the implementation. This straightforward way of generation will result in highly redundant test suites. In many cases, a single test case may satisfy more than one test obligation. Thus, the size of the complete test suite can typically be reduced while preserving coverage.

We use a greedy algorithm to generate reduced test suites. The algorithm operates by randomly picking a test case from the complete test suite, running the test and determining if it improved the overall MC/DC coverage. Any test case that improves the coverage is added to the reduced test set and those that do not are discarded. This is done until we have exhausted all the test cases in the complete test suite. We now—presumably—have a much smaller test suite that achieves the same MC/DC coverage over the implementation. In our experiments we were able to achieve reductions in test suite size of up to 99% while maintaining MC/DC over the implementation. The greedy approach is not guaranteed to be optimal since the order of picking the test cases will affect the size of the reduced set. Nevertheless, creating minimal test suites is not the focus of this paper. We generate three such separate reduced test suites for each full test-suite to decrease the chances of skewing our results with an outlier (very good or very bad reduced test suite).

Experimental Results

For each case example, we generated three reduced non-inlined test suites, three reduced inlined test suites, and three sets of mutants. We ran every test suite against every set of mutants, and recorded the percentage of mutants caught using both the output oracle and the IV oracle. This produced 36 observations for each case example (18 each for the inlined and non-inlined test suites) for use in determining the relative effectiveness of test suites generated from inlined and non-inlined implementations. For each case example, we average the percentage of mutants caught across the different reduced test suites and mutant sets for each {inline level, oracle} pair. This gives us four averages (one each for {Inlined, IV}, {Non-Inlined, IV}, {Inlined, Outputs}, {Non-Inlined, Outputs} pairs) for each case example as summarized in Table 3. Additionally, the table also gives the relative improvement in fault finding of inlined test suites over non-inlined test suites. For example, for the DWM_1 model and IV oracle, the non-inlined test suites kill 79.9% of mutants as compared to 87.9% killed by the inlined test suites, giving a relative improvement of 10.0%. When we use the output oracle with the same test suites and mutants, the non-inlined test suites kill 69.1% of the mutants on average, while the inlined test suites kill 82.5% of the mutants, a relative improvement of 19.4%.

It is evident from Table 3 that the improvement in mutants caught by inlined test suits over non-inlined test suites is substantial (ranging from 10% to a staggering 5940%).

Since our experimental observations are drawn from an unknown distribution—and we therefore cannot reasonably fit our data to a theoretical probability distribution—we used the permutation test (a test with no distribution assumptions [10]) for our statistical analysis. Through this analysis (the details omitted because of space constraints) we can support our hypothesis across *all examples* at the 5% significance level (or lower).

This difference in fault finding is worrisome since it strongly indicates that the fault finding ability of an MC/DC test suite is highly correlated with the structure of the implementation from which it is generated.

Table 3 highlights another observation: even under the best of circumstances investigated in this experiment (i.e., IV oracle, inlined test suite) no more than 90.6% of mutants are caught by an MC/DC adequate test suite. There are several reasons for this poor showing (for example, semantically equivalent mutants and fault masking) that will be covered in the Discussion section. Note again, however, that the poor absolute fault finding does not affect our conclusion, as we are judging relative fault finding, rather than absolute fault finding.

Finally, it is worth noting that the oracle used often has a significant effect on the percentage of mutants caught. The most noticeable effect is on the WBS, the percent of mutants killed increases by more than 20 percentage points when the IV oracle is used in place of the outputs only oracle for both inlined and non-inlined test suites. The difference observed came as a surprise to us and emphasizes the importance of oracle selection when assessing effectiveness of test suites.

Threats to Validity

While our results are statistically significant, they are derived from a small set of examples, which poses a threat to the generalization of the results. Nevertheless, we believe that the examples in our experiment are highly representative and our results are generalizable to systems within the same domain

Another issue is the use of Lustre as an implementation language rather than a more common language such as Java or C. Nevertheless, Lustre's decisions are structured identically to those in imperative languages (such as C or Java), we thus do not view this as a serious threat.

Oracle	**IV**			**Outputs Only**		
Inline Level	**Noninlined**	**Inlined**	**Improvement**	**Noninlined**	**Inlined**	**Improvement**
DWM_1	79.9%	87.9%	10.0%	69.1%	82.5%	19.4%
DWM_2	63.7%	86.1%	35.2%	56%	84.6%	51.8%
DWM_3	5.7%	90.6%	1489%	1.6%	90.6%	5940%
Latctl_Batch	69.3%	86.5%	24.8%	60.1%	79.2%	32.9%
Vertmax_Batch	76.7%	85.5%	11.5%	75.9%	84.7%	11.6%

Table 3: Percentage of mutants caught by reduced inlined and non-inlined test suites.

Our fault seeding method seeds one fault per mutant. In practice, implementations are likely to have more than one fault. However, previous studies have shown that mutation testing in which one fault is seeded per mutant draws valid conclusions of fault finding ability [7].

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

Discussion

In all of our case examples, the inlined test suites revealed significantly more faults than the non-inlined test suites. Statistical analysis of the data revealed that inlined test suites had better fault-finding than non-inlined test suites with better than 95% confidence on all the case studies. The relative improvement in fault finding of the inlined over the non-inlined test suites for the different case examples ranged from 10% to 1489% when using the IV oracle and between 11% to 5940% when using the output oracle.

The results in this experiment are not wholly unexpected but confirm suspicions raised in a previous study [5]. Previously, we compared the effectiveness of inlined versus non-inlined test suites by measuring MC/DC achieved over the inlined implementation. We found that the inlined test suites yielded a significant improvement over non-inlined test suites in coverage achieved over all examples. Table 4 shows the relative improvement in MC/DC achieved (from our previous study) versus the relative improvement in fault finding of inlined over non-inlined test suites for all the case examples. As seen, the relative improvements in fault finding closely follow the relative improvements in coverage achieved. The intuition behind this is covered portions of the implementation represent locations where faults can be detected; since inlined test suites achieve significantly better coverage of the implementation, more faults can be detected than with non-inlined test suites. In the following paragraphs, we discuss the observations and/or concerns raised from these results.

	MC/DC Improvement	**Fault Finding Improvement**	
		IV Oracle	**Out. Oracle**
DWM_1	13.7%	10.0%	19.4 %
DWM_2	49.4%	35.2%	51.8%
DWM_3	635.3%	1489%	5940%
Latctl_Batch	13.3%	24.8%	32.9%
Vertmax_Batch	15.7%	11.5%	11.6%

Table 4: Comparison in Relative Improvement of inlined over non-inlined test suites in MC/DC achieved and Fault Finding.

First, we explored why the inlined test suites outperform the non-inlined test suites. As described in [5], non-inlined test suites do not take the effect of intermediate variable masking into account. In other words, the non-inlined test suites do not ensure that values at intermediate variables affect the output. The inlined test suites on the other hand take masking into account since the test suites are generated from an implementation where intermediate variables are inlined. These test suites are therefore more rigorous, both in terms of test suite size and in accounting for masking, and are thus more effective in fault finding.

The wide range in relative fault finding difference over the different case examples can be attributed to the varied nature and implementation of each system. The DWM_3 system, for instance, consists almost entirely of complex Boolean mode logic. Inlining the implementation thus resulted in very complex Boolean expressions in contrast to the simple Boolean expressions in the non-inlined version, a situation the MC/DC coverage criterion was specifically designed to handle well. This accounts for the abnormally high improvements of 1489% and 5940% that were observed over this system.

Second, none of the test suites (neither inlined nor non-inlined) over the different case examples gave us 100% fault finding—in fact, the fault finding was significantly lower than 100% over all the case examples except the DWM_3 system (the inlined test suites provide over 90% fault finding for this example). This was initially puzzling. We expected to fail to detect the semantically equivalent mutants, but we did not expect this many. Nevertheless, as we studied the models closely we identified several possible reasons for the poor absolute fault finding.

Semantically equivalent mutants: As mentioned previously, we do not determine if a seeded fault results in a mutant that is semantically equivalent to the correct implementation (i.e., a fault that *cannot* result in any observable failure). Thus, in each of our case examples, although we seed 200 faults giving 200 mutated implementations, it may be the case that only 180 of the mutations are semantically different from the correct implementation and can therefore be detected. This is a common problem in fault seeding experiments [7, 11].

In industrial size examples it is extraordinarily expensive, time consuming, and—in most cases—infeasible to identify mutations that are semantically equivalent to the correct implementation and exclude them from consideration. Therefore, the fault finding percentage that we give in our experiment results is a conservative estimate, and we expect the actual fault finding for the test suites to be higher if we were to exclude the semantically equivalent mutations.

Faults in uncovered portions of the model: As seen from our results in [5], in all the case examples except DWM_3 and Latctl_Batch, the maximum achievable MC/DC over the implementation is less than 100%. This implies that there are portions of the implementation for which there exist no test case that can provide MC/DC. We term these as the 'uncovered' portion in the implementation. Note here that the test suites we use in our experiment provide maximum achievable MC/DC over their respective implementation. When seeding faults in an implementation, we may seed faults in the uncovered portion. Since no MC/DC test case could be constructed to cover this portion, the test suite will likely miss such faults. In our experiment, we did not attempt to identify the faults that are seeded in the uncovered portion of the implementation. Such a task would be time consuming and difficult since it would require manually examining the implementation, test suites, and mutations.

Delay expressions: All the case examples we used execute on some form of execution cycle; they sample the environment, execute to completion, and then wait until it is time for a new execution cycle. A *delay expression* is one that uses the value of a variable from a previous execution cycle. MC/DC is not defined over such expressions, and a test suite designed to provide MC/DC will not take these expressions into account. This can severely affect the fault finding capability of such test suites since the test obligations generated for MC/DC will not include the delay operator, and this will in turn affect the length of the generated test cases causing them to be shorter than what is needed to have an effect on outputs. Note that this problem is different from semantically equivalent mutants since it is *possible* to reveal the mutation, but with a test case longer than what is necessary to achieve MC/DC.

To illustrate this problem in a C-like implementation language, consider the program fragment in Table 5.

```
bool Delay_Expr(bool in1,in2)
{
   bool pre_var;
   bool var_a = false;
   while(1)
   {
      pre_var = var_a;
      var_a = in1 or in2;
      return pre_var_a;
   }
}
```

MCDC Test Set for (in1, in2):
`TestSet1 = {(TF),(FT),(FF)}`

Table 5 : Sample program fragment that uses variable values from previous step.

In this code, execution steps are represented as loop iterations. Variables that get used in future execution steps are stored away as intermediate variables for use in later loop iterations. These intermediate variables cannot be inlined for the same reason discussed above. `TestSet1` with one step test cases will provide MC/DC of this program fragment, since we only need to exercise the *or* Boolean expression up to MC/DC. If we were to erroneously replace the *or* operator with *and*, or any other Boolean operator, `TestSet1` providing MC/DC would not be able to reveal the fault. This is because the test cases are too short to affect the output (test cases need to be at least 2 steps long to reach the output). Most systems in the domains of vehicle or plant control (such as the avionics domain) are designed to use variable values from previous steps. Thus, generated test cases to provide MC/DC over such systems will often be shorter than needed to allow erroneous state information to propagate to the outputs and will, therefore, have a detrimental effect on their fault finding capability. Based on this observation and our results in Table 3, we believe that delay expressions not being addressed in the definition of the MC/DC metric is a serious concern and can severely affect the effectiveness of the test suites.

Intermediate variable masking: As mentioned previously, generated test suites do not ensure that intermediate variables affect the output. While we expect such masking to occur with non-inlined implementations, masking is also possible with inlined versions of implementations as they are not completely inlined. We may therefore still have some intermediate variables in the inlined implementations that present opportunities for masking. This may reduce fault finding in inlined implementations, albeit to a much lesser degree than in the non-inlined implementations. Table 5 shows a sample C like program with an intermediate variable `no_alarm` that cannot be inlined and can therefore potentially be masked out in a test case.

```
bool Compute(bool in1,in2,in3)
{
   bool no_alarm;
   if (in1 or in2)
      no_alarm = true
   else
      no_alarm = false;
   return (in3 and no_alarm);
}
```

MCDC Test Set for (in1, in2, in3):
TestSet1={**(TFF)**,**(FTF)**,**(**FFT),TTT)}

Table 6: Sample C like inlined program fragment.

`TestSet1` in Table 6 provides MC/DC over the function; the test cases with `in3 = false` (bold faced) contribute towards MC/DC of the `in1 or in2` condition in the if-then-else statement. However, these test cases mask out the effect of the intermediate variable `no_alarm` since `in3 = false`. Suppose the code fragment in Table 6 was faulty, the correct expression should have been `in1` **`and`** `in2` (which was erroneously coded as `in1` **`or`** `in2`). `TestSet1` providing MC/DC would be incapable of revealing this fault, since there would be no change in the observable output. Thus, seeded faults like the one mentioned here cannot be revealed by a test suite achieving MC/DC because of intermediate variable masking. This observation serves as a reminder of our conclusion in the previous experiment that masking is a crucial consideration for generating test suites that are effective in fault finding.

A work around for this problem in our experiment was to use intermediate variables in addition to outputs as part of our oracle comparison (IV oracle). The use of the IV oracle significantly improved the fault finding capability of both the inlined and non-inlined test suites. For the non-inlined test suite, the improvement in fault finding ranged from 0.8% points on the Vertmax_Batch system (75.9% to 76.7%) to a maximum improvement of about 10% points on the DWM_1 system (69.1% to 79.8%). For the inlined test suite, the improvement in fault finding ranged from 0% on the DWM_3 system (constant at 90.6%) to a maximum improvement of about 7% points on the Latctl_Batch system (79.2% to 86.5%). Thus, we find that the oracle selected can have a significant impact on the fault finding effectiveness of test suites. The oracle issue merits further discussion and study, but it is not the focus of this paper and we hope to look at it in more detail in our future work.

Conclusion

Our empirical investigation in this paper revealed that test suites generated to provide MC/DC are highly to the structure of the implementation they are designed to cover. Our data revealed that our hypothesis stating that test suites generated to be MC/DC adequate on an inlined implementation have greater fault finding ability than test suites generated to be MC/DC adequate on the non-inlined version of the same implementation. This observation confirms our suspicion that MC/DC used as a test adequacy metric is highly sensitive to structural changes in the implementation, and that test suite adequacy measurement using the MC/DC metric will be better served if done over the inlined implementation. Unfortunately, as far as we are aware, there is no discussion of this issue in current and evolving standards, and we strongly encourage the organizations contemplating using structural coverage criteria to assess the adequacy of a testing effort to consider our findings.

Delay expressions (expressions that refer to variable values in previous execution steps) are currently not considered in the definition of MC/DC, and we believe that test cases generated to provide MC/DC that do not take these expressions into account will be less effective in fault finding since the generated test cases may be shorter than needed to affect the outputs. We recommend considering delay expressions in future definitions of MC/DC.

Finally, the oracle used can have a substantial effect on the fault finding ability. We found that using internal variables in addition to outputs in our oracle provided significantly better fault finding than using only outputs. This oracle issue has serious implications regarding the adequacy of testing efforts based on structural coverage. A statement such as *"We have completed testing up to Masking MC/DC and revealed no critical faults"* has little meaning unless there is a discussion about the nature of the oracle used. If the output variables only are used in the oracle, it is—from our observations in this paper—highly likely that you may have *encountered* faults but failed to *reveal* them. The oracle selection problem and its implication in fault finding merits more attention and we hope to take a closer look at this issue it in our future work.

References

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

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

[3] Mathworks Inc. Simulink product web site. Via the world-wide-web: www.mathworks.com /products /simulink.

[4] Esterel-Technologies. SCADE Suite product description. http://www.esterel-technologies.com /v2/scadeSuiteForSafetyCriticalSoftwareDevelopment/index.html, 2004.

[5] Ajitha Rajan, Michael Whalen, and Mats Heimdahl. The Effect of Program and Model Structure on MC/DC Test Adequacy Coverage. In *Proceedings of 30th International Conference on Software Engineering (ICSE)*, May 2008.

[6] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[7] J.H. Andrews, LC Briand, and Y. Labiche. Is Mutation an Appropriate Tool for Testing Experiments? *Proceedings of the 27th international conference on Software engineering*, pages 402–411, 2005.

[8] A. Memon, I. Banerjee, and A. Nagarajan. What test oracle should I use for effective GUI testing? *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 164–173, 2003.

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

[10] J.P. Shaffer. Multiple Hypothesis Testing. *Annual Review of Psychology*, 46(1):561–584, 1995.

[11] A.J. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification & Reliability*, 7(3):165–192, 1997.

Acknowledgements

We would like to thank Dr. Elizabeth Whalen from Boeing Co. for her help with the statistical analysis of the results, and John Chilenski from Boeing Co. for his insights and discussions regarding the pitfalls of structural coverage metrics.

This work has been partially supported by NASA Ames Research Center Cooperative Agreement NNA06CB21A, NASA IV&V Facility Contract NNG-05CB16C, and the L-3 Titan Group.