Distinguished Paper

The Effect of Program and Model Structure on MC/DC Test Adequacy Coverage*

Ajitha Rajan
Dept. of Comp. Sci. and Eng.
University of Minnesota
arajan@cs.umn.edu

Michael W. Whalen
Advanced Technology Center
Rockwell Collins Inc.
mwwhalen@rockwellcollins.com

Mats P.E. Heimdahl
Dept. of Comp. Sci. and Eng.
University of Minnesota
heimdahl@cs.umn.edu

ABSTRACT

In avionics and other critical systems domains, adequacy of test suites is currently measured using the MC/DC metric on source code (or on a model in model-based development). We believe that the rigor of the MC/DC metric is highly sensitive to the structure of the implementation and can therefore be misleading as a test adequacy criterion. We investigate this hypothesis by empirically studying the effect of program structure on MC/DC coverage.

To perform this investigation, we use six realistic systems from the civil avionics domain and two toy examples. For each of these systems, we use two versions of their implementation—with and without expression folding (i.e., inlining). To assess the sensitivity of MC/DC to program structure, we first generate test suites that satisfy MC/DC over a non-inlined implementation. We then run the generated test suites over the inlined implementation and measure MC/DC achieved. For our realistic examples, the test suites yield an average reduction of 29.5% in MC/DC achieved over the inlined implementations at 5% statistical significance level.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing Tools*

General Terms

Experimentation, Verification

*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.

ICSE'08, May 10–18, 2008, Leipzig, Germany.

1. 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. Such criteria can be useful tools when evaluating a testing effort. 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. In our work we have been particularly interested in the coverage criterion Modified Condition and Decision Coverage (MC/DC) [2] since it is used as an exit criterion when testing software for highly critical software in the avionics industry. For certification of such software, a vendor must demonstrate that the test-suite provides MC/DC coverage of the source code [20]. In addition, the increased use of model-based development (MBD) using tools such as Simulink [13] and SCADE [4] 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 [22].

Our concern regarding MC/DC is its sensitivity to the structure of the program or model under test. A straightforward way to reduce the difficulty of achieving MC/DC coverage over a program is to introduce additional variables to factor complex Boolean expressions into simpler expressions. In this paper, we examine the effect of such transformations by comparing test suites necessary to cover programs consisting of simple decisions (consisting of at most one logical operator) versus programs in which such intermediate variables are removed. We refer to these versions as the non-inlined and inlined implementations, respectively. In our experiment we found the effect of such transformations to be dramatic: in one case, a suite that achieved 100% coverage of the non-inlined implementation yielded only 13.6% coverage of the inlined implementation. Our analysis revealed that the transformations yield an average reduction of 29.5% in MC/DC measured over inlined implementations (which was statistically significant for a null hypothesis of no difference at the 5% significance level). Many of the discrepancies are due to situations when the effect of the code structure being tested is 'masked out' by another condition, and cannot affect the result computed by the implementation. Given that the inlined and non-inlined implementations are

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

semantically equivalent, the discrepancy in the rigor of the testing metric under this simple transformation is a cause for concern.

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 outcomes 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). As an example, consider the trivial program fragments in Table 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.

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, the correct expression 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.

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.

The issue with decision structure is not confined only to code. The move towards model-based development in the critical systems community makes test-adequacy measurement a crucial issue in the modeling domain. Commercial modeling notations such as Simulink [13] and SCADE [4] are gaining widespread acceptance. 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

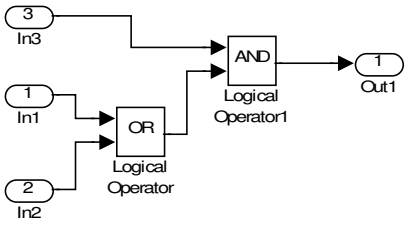

Figure 1: Simulink model of example in Table 1

'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. Test-suites designed to provide MC/DC coverage over the models could, therefore, be potentially ineffective. Thus, the current approach to measuring MC/DC over such models is 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.

In this paper, we empirically study the effect of inlining on MC/DC achieved. For our case study we used six systems from the civil avionics domain and two toy examples. For each of these examples we first generated test suites that provide MC/DC over an implementation with no inlining. We then ran the test suite over an implementation where temporary variables had been inlined, and measured MC/DC achieved. This measurement was then compared against the achievable coverage of the inlined program to assess the sensitivity of the MC/DC metric to inlining.

Our case studies revealed that inlining the implementation has a profound effect on the MC/DC achieved by a test suite. For all six industrial examples in our case study, a test suite providing MC/DC over the non-inlined version achieved significantly lower coverage on the inlined version. Coverage reductions ranged from 12% - 86% (of the achievable coverage) on these examples. Needless to say, all uncovered portions of the programs represent locations where faults of the type discussed earlier in this section may reside undetected. On the other hand, for the toy examples, negligible reduction (less than 1%) in MC/DC was observed. Statistical analysis of these results revealed that for industrial systems, our hypothesis that the MC/DC metric is sensitive to the structure of the implementation is supported.

We find the effect of program or model structure on MC/DC coverage as well as the general lack of awareness of the (potential) problems worrisome; in particular with respect to the efficacy of test suites developed to provide model coverage. Engineers and certifiers should be aware and cautious of this issue when using MC/DC to assess the adequacy of test suites for safety-critical applications. We are aware that test suite assessment using coverage is not adequate and only serves as an initial indication; the real concern is in their fault finding capability. In the future, we plan to further evaluate this hypothesis by running fault-finding experiments.

The remainder of the paper is organized as follows. Section 2 introduces our experimental setup and the case examples used in our investigation. Techniques used to inline the implementation are described in Section 2.3. We discuss our test case generation methodology in Section 2.4. Section 2.5 describes the coverage measurements gathered over

the implementations. Results obtained and their analysis is presented in Section 3. Finally, Section 4 discusses the implications of our results, and points to future directions in evaluating the sensitivity of the MC/DC metric.

2. EXPERIMENT

To investigate how MC/DC is affected by the structure of a implementation we designed our experiment to test the following hypothesis:

Hypothesis: A test-suite generated to provide MC/DC over the non-inlined implementation will achieve lower MC/DC over an implementation that is inlined.

We formulated our hypothesis based on the observation that the effect of masking may not be revealed in the non-inlined version of the implementation and, therefore, it may be significantly easier to achieve MC/DC than on the inlined version that forces the construction of test cases to take masking into account. To illustrate this idea consider again the example in Table 1. In the two behaviorally equivalent versions of the implementation for `out_1`, TestSet1 will achieve 100% MC/DC over Version 1 because the masking effect of `in_3` is not revealed (MC/DC coverage of `stmt1` is credited even though the effect of `expr_1` is masked out in `stmt2`). Nevertheless, the test-set will only achieve 33% MC/DC over the inlined version (2 of 3 cases for 'and', and 0 of 3 cases for 'or') since `in_3 = false` masks out the effect of `in1 or in2`[1].

Chilenski investigated three different notions of MC/DC [1], namely: Unique Cause MC/DC, Unique Cause + Masking MC/DC, and Masking MC/DC. In this paper we use *masking* MC/DC [8] to determine the independence of conditions within a Boolean expression. In masking MC/DC, a basic condition is *masked* if varying its value cannot affect the outcome of a decision due to structure of the decision and the value of other conditions. To satisfy masking MC/DC for a basic condition, we must have test states in which the condition is not masked and takes on both *true* and $false$ values. Masking MC/DC is the easiest of the three forms of MC/DC to satisfy since it allows for more independence pairs per condition and more coverage test sets per expression than the other forms. Chilenski's analysis showed that even though Masking MC/DC could allow fewer tests than Unique-Cause MC/DC, its performance in terms of probability of error detection was nearly identical to the other forms of MC/DC. This led Chilenski to conclude in [1] that Masking MC/DC should be the preferred form of MC/DC.

To better illustrate the definition of masking MC/DC, consider the expression *A and B*. To show the independence of B, we must hold the value of A to *true*; otherwise varying B will not affect the outcome of the expression. Independence of A is shown in a similar manner. Table 2 shows the test suite required to satisfy MC/DC for the expression *A and B*. When we consider decisions with multiple Boolean operators, we must ensure that the test results for one operator are not masked out by the behavior of other operators. For example, given *A or* (*B and C*) the tests for *B and C* will not affect the outcome of the decision if

[1]Note here that there is no standard way of measuring partial MC/DC and others might come up with a different number. This is a topic for a separate report, however.

A	B	*A and B*
T	T	T
T	F	F
F	T	F

Table 2: Example of a test suite that provides Masking MC/DC over *A and B*

A is *true*. Table 3 gives one of the test suites that would satisfy masking MC/DC for the expression *A or* (*B and C*). Note that in Table 3, test cases {2,4} or {3,4} will demonstrate independence of condition A under Masking MC/DC. Nevertheless, these test cases will not be sufficient to show independence of A under Unique Cause MC/DC since the values for conditions B and C are not fixed between the test cases.

A	B	C	*A or (B and C)*
F	T	T	T
F	T	F	F
F	F	T	F
T	F	F	T

Table 3: Example of a test suite that provides Masking MC/DC over *A or* (*B and C*)

2.1 Experimental Setup

To evaluate our hypothesis, we generate a test-suite to provide MC/DC over a non-inlined version of the implementation, and run the test suite over the inlined version of the same implementation and measure MC/DC achieved; we are then in a position to pass judgement on whether or not a program where complex decisions are broken down into many small conditions reduces the effectiveness of the MC/DC coverage criterion. Subsequent paragraphs provide more information on the experimental procedure.

To provide realistic results, we conducted experiments on six industry sized examples: three models from a display window manager for an air-transport-class aircraft (DWM_1, DWM_2, and DWM_3), and three models related to flight guidance mode logic (ToyFGS_05, Vertmax_Batch and Latctl_Batch). We also conducted experiments using two toy case examples: a Wheel Brake System, and a Sensor Voting model developed at Honeywell Labs. Section 2.2 gives a description of each of the models used in our experiments. For each of the case examples, we conducted the experiment following the steps described below.

1. From the specification of the 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. Section 2.3 provides more information on the inlining mechanism used in our experiment. We will refer to the implementation generated in this manner as the 'inlined' version.

2. From the non-inlined version of the implementation we generated a test-suite that provided maximal MC/DC coverage. The test suites were naïvely generated, which led to much larger suites than actually necessary for maximal coverage. It is worth noting that the generated test suite may not always provide 100% MC/DC coverage over the implementation since some constructs may not be reachable. Thus, 'achievable' MC/DC may be lower than 100%; our generated test-suites provided achievable MC/DC coverage over the non-inlined models. It was proven (by model checking) that our suites provide maximal achievable coverage.

3. We performed a naïve reduction of the test suite generated in Step 2 ensuring the MC/DC coverage over the non-inlined version was maintained (a greedy algorithm).

4. We ran the reduced test suite over the inlined version of the implementation, and measured the MC/DC coverage achieved.

5. Finally, we compared the result from Step 4 with 'achievable coverage' on the inlined implementation to determine how much (if any) of the inlined implementation was not covered by the tests providing maximal coverage of the non-inlined implementation.

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

2.2 Case Examples

In our experiment, we used six close to production or production systems and two toy examples. All systems used in our experiment were modeled using the Simulink [13] notation from Mathworks Inc. We provide descriptions for these systems below.

2.2.1 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 three FGS models in this paper focus on the mode logic of the FGS. The first model (ToyFGS_05) is a Simulink version of the prototype model described in [14]. The Vertmax_Batch and Latctl_Batch models describe the vertical and lateral mode logic for the Rockwell Collins FCS 5000 flight guidance system family, described in [15].

2.2.2 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). It also must manage the cursor, ensuring that the cursor does not appear on a display that contains an application that does not support the cursor. In the event of reversion, the DWM must ensure that the cursor is not tasked to a dead display.

2.2.3 Sensor Voting Model

The triplex sensor voter used in our experiment is a slightly simplified version of one developed at Honeywell Laboratories. The design is that of a generic triplex voter: the voter takes inputs from three redundant sensors and synthesizes a single reliable sensor output. Each of the redundant sensors produces both a measured data value and self-check bit (validity flag) indicating whether or not the sensor considers itself to be operational. All valid sensor signals are combined to produce the voter output. If three sensors are available, a weighted average is used in which an outlying sensor value is given less weight than those that are in closer agreement. If only two sensors are available a simple average is used. If only one sensor is available, it becomes the output. There are two mechanisms whereby a faulty sensor may be detected and eliminated: comparison of the redundant sensor signals and monitoring of the validity flags produced by the sensors themselves.

2.2.4 Wheel Brake System (WBS)

The Wheel Brake System (WBS) is a Simulink model derived from the WBS case example found in ARP 4761 [21, 10]. The WBS is installed on the two main landing gears. Braking on the main gear wheels is used to provide safe retardation of the aircraft during the taxiing and landing phases, and in the event of a rejected take-off. Braking on the ground is either commanded manually, via brake pedals, or automatically (autobrake) without the need for pedal application. The Autobrake function allows the pilot to prearm the deceleration rate prior to takeoff or landing. When the wheels have traction, the autobrake function will control break pressure to provide a smooth and constant deceleration.

2.3 Inlined and Non-Inlined Implementations

The case examples described in Section 2.2 were modeled in Simulink and we 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 [7]. Lustre is a synchronous dataflow language and is the underlying notation for the SCADE Suite from Esterel Technologies [3]. We translated each of the case examples

Non-Inlined Implementation:

```
node Compute(x,y,z: bool;
             temp,thresh: int)
    returns(out: bool);
 var
    run, no_danger, no_alarm: bool;
 let
    run = AndOr(x,y,z);
    no_danger = (temp <= thresh);
    no_alarm = if (no_danger)
                then true
                else false;
    out = run and no_alarm;
 tel;

node AndOr(a,b,c: bool)
    returns(out: bool);
 var
    local: bool;
 let
    local = b or c;
    out = a and local;
 tel;
```

Inlined Implementation:

```
node Compute(x,y,z: bool;
             temp,thresh: int)
    returns(out: bool);
 var
    no_alarm: bool;
 let
    no_alarm = if (temp <= thresh)
                then true
                else false;
    out = (x and (y or z)) and no_alarm;
 tel;
```

Table 4: Example implementation with and without inlining

modeled in Simulink to Lustre. We 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 [12], 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.

2.3.1 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 Lustre implementation with no inlining is presented in Table 4. The implementation shown in the table, 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.

2.3.2 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 4 presents the inlined version of the previously mentioned example.

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.

2.4 Test Suite Generation and Reduction

Several research efforts [18, 19, 6] have developed techniques to automatically generate test cases using model checkers. We use the same techniques to generate test suites that provide MC/DC over the implementation. We used the NuSMV [17] model checker in our experiments. The steps followed in test suite generation were as follows:

1. The Lustre implementation was translated into the input language of NuSMV [17] using the translation infrastructure developed in a previous project [23].
2. We augmented the translation infrastructure with the capability to automatically generate MC/DC obligations for constructs in the implementation. Using this capability, we generated the requisite MC/DC obligations and negated them to form trap properties [6] for the model checker.
3. We merged the NuSMV model in step 1 along with the trap properties in step 2 and gave it to the model checker which then generates counter examples that constitute the test suite providing MC/DC coverage over the Lustre implementation in step 1.

Note that not every trap property generated in step 2 will result in a test case (counter-example), since some parts of the model may be unreachable or specific combinations of truth values infeasible (for example, because of masking). Therefore, expressions occurring in those portions of the model may not be exercised up to MC/DC, and the 'achievable' MC/DC over an implementation with such expressions will be less than a 100%. To assess the achievable MC/DC over the implementation, we run the generated test suite and *measure* MC/DC achieved over the implementation. Section 2.5 provides more information on the coverage measurement tool. We use this tool to measure 'achievable MC/DC' over the inlined and non-inlined implementation.

We attempt to generate a separate test case for every construct we need to cover in the implementation. This is a straightforward way of generating test suites but the suites will be highly redundant. In many cases, a single test case may satisfy more than one test obligation. For instance, a

test case designed to cover a certain MC/DC truth assignment of one decision will most likely also cover many other truth assignments on other decisions. Thus, the size of the complete test suite can typically be reduced while preserving coverage.

Since in this experiment we are interested in how implementation structure affects MC/DC coverage, we are interested in a small test suite so that redundant test cases do not artificially inflate the coverage measured on the inlined implementation. Therefore, in our experiments, we performed test suite reduction and we used a simple greedy method. We begin with an empty set of test cases and initialize the coverage to zero. The greedy algorithm then sequentially picks a test case from the complete test suite, runs the test, and determines if the test case improved the overall MC/DC achieved. Any test case that improves the coverage achieved 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 naïve greedy manner is not guaranteed to be optimal, since the order of picking the test cases will affect the size of the reduced set. Since the experiments in this paper did not require large test suites to provide MC/DC coverage, we did not investigate creating minimal test suite sets (although it would be desirable to have this minimal set).

2.5 Measuring MC/DC Coverage

For this project we built a test-adequacy measurement tool for Lustre that takes a test suite and an implementation (in Lustre), and then gathers information on the MC/DC coverage achieved over the implementation. We currently measure masking MC/DC [8]. In our case studies, we used the coverage measurement tool to gather MC/DC measures over the inlined and non-inlined implementations (recall that Section 2.3 provides a description of the terms inlined and non-inlined) using different test suites. We gathered the following metrics in our experiment:

Measured MC/DC Coverage: Here we simply run a test-suite and measure the MC/DC coverage achieved over the implementation.

Achievable MC/DC Coverage: As a result of masking and unreachability mentioned in Section 2.4, a MC/DC test suite generated from an implementation may not provide 100% MC/DC coverage. We determine the achievable MC/DC for the implementation by (1) generating MC/DC test suite using our test-case generation tool (this will find all feasible test cases), and (2) using the generated test suite we execute all tests and measure the achievable MC/DC (the MC/DC achieved by the feasible tests).

Given these two measures we can investigate the effect of inlining on the MC/DC coverage of an implementation.

3. EXPERIMENTAL RESULTS

For every case example described in Section 2.2, we generated a non-inlined implementation as well as an inlined implementation.

Non-inlined Implementation: For the non-inlined implementation, we measured the following:

1. **Achievable MC/DC Coverage,**
2. **Size of the complete test suite** generated to provide MC/DC over the non-inlined implementation, and
3. **Size of the reduced test suite** that maintains MC/DC achieved.

Inlined Implementation: For the inlined implementation, we measured the following:

1. **Measured MC/DC Coverage** achieved by running the reduced test suite that was generated from the non-inlined implementation,
2. **Achievable MC/DC Coverage,**
3. **% of Achievable Coverage** which is the ratio of Measured MC/DC over Achievable MC/DC on the inlined implementation,
4. **Size of the complete test suite** generated from the *inlined implementation* enabling us to establish the achievable MC/DC of the inlined implementation, and
5. **Size of the reduced test suite** that maintains Achievable MC/DC.

Tables 5 and 6 summarize the above measures for the different case examples.

In Table 5, the full test suite represents the number of *feasible* MC/DC obligations in the non-inlined implementations for which we generated a test case, for example, ToyFGS_05 has 4,445 feasible MC/DC obligations that must be covered and we generate one test case for each one. The reduced set indicates the number of test cases our simple greedy reduction algorithm generated to provide the same coverage as the full set. For ToyFGS_05, 75 test cases were sufficient to provide the maximum achievable coverage. This reduced set column is repeated in Table 6 since we run the reduced test sets on the inlined implementations.

With these test-suites we are in a position to measure the coverage of the original non-inlined implementations as well as the inlined implementations (Tables 5 and 6). Note here that the test suites for the non-inlined implementations provide 100% achievable coverage for all non-inlined implementations (column labelled **%Achievable** in Table 5).

We then ran the reduced tests sets on the inlined implementations and measured the coverage (Table 6). At first glance the MC/DC coverage numbers are alarming; coverage dropped dramatically in most cases (Measured MC/DC in Table 6). This number is misleading, however. The *measured coverage* must be judged in the context of *achievable coverage*; when decisions are inlined to form more complex conditions the effect of masking and infeasible combinations of conditions will become visible and the achievable coverage typically drops. We generated and ran test suites that provided 100% achievable coverage over the inlined implementations to determine the maximum possible coverage (Achievable MC/DC in Table 6). We can now see that the drop in MC/DC coverage is not quite as dramatic as indicated by the raw Measured Coverage numbers—the % Achievable

	# Tests		Achievable MC/DC	% Achievable
	Full	Reduced		
ToyFGS_05	4445	75	91.2%	100%
DWM_1	180	18	95.1%	100%
DWM_2	299	39	96.2%	100%
DWM_3	2522	23	100%	100%
Latctl_Batch	315	52	98.9%	100%
Vertmax_Batch	1415	235	100%	100%
WBS	271	10	76.6%	100%
Sensor Voting Model	103	10	68.7%	100%

Table 5: MC/DC coverage achieved over the non-inlined implementations.

	# Tests Reduced	Measured MC/DC	Achievable MC/DC	% Achievable
ToyFGS_05	75	33.6%	41.9%	80.2
DWM_1	18	81.2%	92.3%	87.9
DWM_2	39	61.9%	92.5%	66.9
DWM_3	23	13.6%	100%	13.6
Latctl_Batch	52	88.3%	100%	88.3
Vertmax_Batch	235	86.2%	99.8%	86.3
WBS	10	77.7%	77.7%	100
Sensor Voting Model	10	56.4%	57.2%	98.6

Table 6: MC/DC coverage achieved over inlined implementations.

column points out the percentage of the achievable MC/DC obligations that were actually covered by the test-suite—but they are still a serious concern. For the six industrial sized models the test-suites providing 100% of the achievable coverage of the non-inlined implementations provided between 13.6% and 88.3% of the achievable coverage on the inlined implementations. This is clearly worrisome since the inlined implementation is semantically equivalent to the non-inlined implementation and all uncovered MC/DC obligations could potentially contain a fault.

The wide range in coverage reductions observed over the different systems was due to their varied behavior and implementations. The DWM_3 system, which recorded the maximum reduction, consists almost entirely of complex Boolean mode logic. Inlining the implementation thus resulted in complex Boolean expressions in contrast to the very simple Boolean expressions in the non-inlined version. On the other hand, the Latctl_Batch system, with the least coverage reduction among the industry sized examples, was primarily a functional transform with less complex mode logic and inlining did not have a dramatic effect on the complexity of Boolean expressions. Therefore, the mismatch between the measured MC/DC and the achievable MC/DC was not as dramatic on this system. We provide statistical analysis of the results from the industrial and toy examples in Section 3.1.

The very low achievable coverage (only about 41.9% of the MC/DC obligations can actually be covered) of the ToyFGS_05 example merits a closer examination. Low achievable coverage is an indication of decisions where large parts are masked out due to lazy evaluation or the control flow of the implementation; either way, it indicates a large number of conditions that simply have no impact on the program behavior. Low achievable coverage is an issue that should cause concern and spur investigation. In this case, the Simulink version of the ToyFGS_05 case-example we used in this study was a literal translation of a previous version expressed in the RSML^{-e} notation [16]. Boolean expressions in RSML^{-e} are expressed as And/Or tables [11] that are basically nicely formatted Boolean expressions in disjunctive normal form. Therefore, the decisions are often large and contain a lot of redundancy which creates significant opportunities for masking. As a consequence, there are many MC/DC combinations of truth values that are simply not feasible.

Finally, let us study the size of the test-suites needed to provide coverage over the various implementations. Table 7 gives the sizes of test suites generated to provide MC/DC over the non-inlined implementations versus the inlined implementations. (Note again that the full test-suites represent one test-case per *achievable* MC/DC obligation.) As seen in the table, there is no regular trend in the number of achievable MC/DC obligations for the two versions; the number of obligations to cover may go up or down. Nevertheless, the reduced test-suites for the inlined implementation are always as big or bigger (1 to 20 times for our examples) than the non-inlined counterpart. If we inline we will get fewer decisions, but they will be larger and—in general—more difficult to cover with a test-case. Therefore based on the results in our experiment, to cover an inlined implementation we need *a greater number and more rigorously prepared* test-cases to achieve the desired coverage.

3.1 Statistical Analysis and Evaluation of Hypothesis

In this section, we analyze the results in Tables 5 and 6 and determine if the hypothesis stated in Section 2 that *"a test suite generated to provide MC/DC over the non-inlined*

	Non-Inlined Implementation		Inlined Implementation	
	# Tests		# Tests	
	Full	Reduced	Full	Reduced
ToyFGS_05	4445	75	1909	166
DWM_1	180	18	121	29
DWM_2	299	39	946	88
DWM_3	2522	23	2697	463
Latctl_Batch	315	52	205	77
Vertmax_Batch	1415	235	1464	285
WBS	271	10	125	10
Sensor Voting Model	103	10	189	12

Table 7: Sizes of Test Suites providing MC/DC over implementations with and without Inlining.

implementation will achieve lower MC/DC coverage over the inlined implementation" is supported. From here on, we will refer to this hypothesis as H. To evaluate H, we first formulate our null hypothesis $H0$ as follows:

$H0$**:** A test suite generated to provide achievable MC/DC over the non-inlined implementation will provide *achievable* MC/DC over the inlined implementation.

Thus, to accept H we would have to reject the null hypothesis $H0$. We are aware that the number of samples used in our experiment is rather small, and would therefore be unreasonable to fit the data to a theoretical probability distribution. We therefore test the hypothesis by not assuming any particular distribution. To do this, we use the permutation test, a non-parametric test with no distributional assumptions, to evaluate our hypothesis. A permutation test is a type of statistical significance test in which a reference distribution is obtained by calculating all possible values of the test statistic under rearrangements of the labels on the observed data points [5]. Note that the permutation test presented here is only for the results from the six industrial systems. A discussion of the results from the toy examples and why they were excluded from our analysis is presented at the end of this section. To perform the permutation test, we restate the null hypothesis as:

$H0$**:** The %Achievable data from the inlined and non-inlined implementations come from the same population.

In our experiment we have two groups of data – non-inlined % Achievable (group A), and inlined % Achievable (group B) – that are paired (100 paired with 80.2, 100 paired with 87.9, 100 paired with 66.9, 100 paired with 13.6, 100 paired with 88.3 and 100 paired with 86.3). Group A has all values of 100 and a sample mean $\mu_A(= 100)$ and group B has the other values (80.2, 87.9, 66.9, 13.6, 88.3, 86.3) with a sample mean $\mu_B(= 70.5)$. We want to test at 5% significance level ($\alpha = 0.05$) whether they come from the same population (the null hypothesis). If they do then the paired values can be switched, for instance 80.2 could occur in group A and its pair value of 100 could occur in group B. The two-sided permutation test is designed to determine whether the observed difference between the sample means is large enough to reject the null hypothesis $H0$ that the two groups have identical probability distribution.

The test proceeds as follows. First, the absolute value of the difference in means between the two samples is calculated - this is the observed value of the test statistic, $T(stat)$.

$$T(stat) = abs(\mu_A - \mu_B) = 100 - 70.5 = 29.5$$

We then calculate the number of ways of grouping the paired values into 2 sets. With a sample size of 6 paired values,

$$Number\ of\ Permutations = 2^6 = 64$$

Let $COUNT$ be the number of permutations with absolute difference in means greater than or equal to the observed value, $T(stat)$. For our sample data, we found $COUNT = 2$ (One permutation where group A had all 100s and group B had the other values, and another permutation where group B had all 100s and group A had the other values).
The two-sided $P - Value$ for the test is calculated as:

$$P - Value = COUNT\ /\ Number\ of\ Permutations$$
$$= 2/64 = 0.031$$

Since our $P - Value$ (0.031) is less than the α value (0.05), for the industrial models the null hypothesis $H0$ that the %Achievable data from the inlined and non-inlined implementations come from the same population would be rejected at the level of significance $\alpha = 0.05$. Based on these results, the alternate hypothesis that states % Achievable MC/DC for the inlined and non-inlined models come from different populations is supported. Since the %Achievable MC/DC for the non-inlined model is always at 100, %Achievable inlined MC/DC can only be lower than %Achievable non-inlined. Thus, for the industrial models in our experiment, our hypothesis (H) that states a test suite generated to provide MC/DC over the non-inlined implementation will achieve lower MC/DC coverage over the inlined implementation is *supported*.

For the toy examples in our case study (WBS and Sensor Voter) the generated test suites provided close to achievable MC/DC on the inlined implementation. This was somewhat surprising to us since these results differ significantly from our other data points. The results can be attributed to the simple Boolean decisions typically present in toy examples; decisions that are largely unaffected by inlining. We do not believe these negative results affect the support of our hypothesis (When the toy examples are included in the analysis, the null hypothesis is still rejected but the average reduction in coverage is smaller). Instead, these results should act as a warning that experimenting with toy examples can be misleading—experimentation must take place with real or industrial systems exhibiting both the *size and structure* seen in practice.

3.2 Threats to Validity

We see two threats to the external validity of our experiment—the selection of case examples and the use of Lustre as an implementation langauge.

Although our results are statistically significant, to generalize them across the class of systems where MC/DC coverage may be of interest, it would be desirable to select more case examples than the six we have selected. We believe, however, that the examples we used are highly representative and our results are generalizable to other systems in the same domain.

We used Lustre as an implementation language in this study rather than a commonly used langauge such as C or C++. We do not see this as a serious problem since the structure of decisions is identical to traditional imperative languages. Therefore, we believe our results are generalizable to a 'normal' implementation.

The threat to internal validity in our experiment is the naïve test suite reduction algorithm used—the algorithm will give us a reduced test-suite, but there is no guarantee that it will be the smallest (or even small). Had we been able to obtain the minimal set of test-cases (an NP-complete problem) the effect observed in our experiments would—we believe—have been even more dramatic. Therefore, we do view our naïve reduction algorithm as a problem.

4. DISCUSSION AND CONCLUSIONS

There are three primary conclusions that can be derived from the results in Tables 5 through 7.

Firstly, for our six industrial examples, test suites that provide MC/DC on the non-inlined implementation did poorly on the inlined implementations. The reason for this result is that MC/DC measurement on the former does not take the effect of masking into account while measuring over the latter does. Masking is a *crucial* consideration for generating test suites that are rigorous and effective in fault finding. Keeping this in mind, we believe there is a serious need for one of the following:

1. a coverage metric (or test adequacy metric) that takes masking into consideration irrespective of implementation structure, or

2. a canonical way of structuring code so that condition masking is revealed when measuring coverage using existing coverage criteria.

This observation takes on an additional dimension of importance in the model-based domain since there have been suggestions to allow measurement of test adequacy in the model domain as a substitute for measurement in the code domain. The reason being the instrumentation necessary to measure coverage on the code often leads to a very costly testing process (it is not only expensive to *find* the tests, but also to *run* the tests) as opposed to measuring coverage on the model where instrumentation is much easier. Modeling languages such as Simulink and SCADE are extensively used in the critical systems domain and current coverage metrics for these languages provide the weakest possible MC/DC obligation; all decisions are simple (for example, only one *or*, *and*, or *implication* in each decision). Therefore, measuring coverage over these models as opposed to the code derived from the models exposes the problems discussed in this paper and would, in our opinion, have the potential to significantly weaken existing MC/DC test obligations. Based on our results in this experiment, we advise that new MC/DC coverage metrics be proposed that account for condition masking or to restructure the implementation (for e.g., inlining) so that masking is revealed.

Secondly, in addition to judging quality of test suites, measuring coverage over the inlined implementation will help developers better assess the independence of conditions, i.e., do the conditions affect the outcome of execution, within an application. With a non-inlined implementation, coverage results may be highly misleading, as seen for the FGS system in Table 6. The achievable MC/DC over the non-inlined implementation of the FGS was 91.2% as compared to an achievable MC/DC of 41.9% over the inlined implementation. The difference in the numbers is unnerving since these additional conditions contributing to coverage in the non-inlined implementation are essentially 'dead code' in the sense that they cannot affect the outcome of the execution. Based on these results, we believe reporting coverage numbers for the non-inlined implementation may hide many instances of useless code.

Finally, as Table 7 illustrates, for all case examples the size of the reduced test suite that provides MC/DC over the inlined implementation is larger than that for the non-inlined implementation. Although finding more test cases is an added burden, previous research efforts [9] have shown that the larger test suite size necessitated by the inlined-implementation will enhance its fault finding effectiveness.

To summarize, our empirical investigation in this paper revealed that for all six industrial examples the MC/DC metric was highly sensitive to the structure of the implementation. Statistical analysis on the experimental data revealed that our hypothesis stating that test suites generated to be MC/DC adequate on a non-inlined implementation is *inadequate* on the inlined version of the same implementation was supported at 5% significance level. Inadequacy ranged from 12 to 86 percentage points for the different systems. Thus, we conclude that the MC/DC metric can be highly subjective as a test adequacy metric. In the absence of a metric that is robust to structural changes in the implementation, we believe that test suite adequacy measurement using the MC/DC metric will be better served if done over the inlined implementation. The reason being the inlined implementation requires a more rigorous test suite that we postulate will provide better fault finding and better assess whether masked conditions exist in critical software.

In future work we will further evaluate the sensitivity of MC/DC to code structure with fault finding experiments. We also plan to investigate canonical forms of the MC/DC metric that are more robust to structural changes in the implementation.

5. ACKNOWLEDGEMENTS

We would like to thank Dr. Elizabeth Whalen from Boeing Co. for her help with the statistical analysis of the presented results. We would also like to thank John Chilenski from Boeing Co. for his insights and discussions about the pitfalls of structural coverage metrics. We thank Dr. Steve Miller and Dr. Alan Tribble of Rockwell Collins Inc. for their help and support with the flight control systems.

6. REFERENCES

[1] J. Chilenski. An investigation of three forms of the modified condition decision coverage (mcdc) criterion. Technical Report DOT/FAA/AR-01/18, Office of Aviation Research, Washington, D.C., April 2001.

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

[3] Esterel-Technologies. Corporate web page. www.esterel-technologies.com, 2004.

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

[5] R. Fisher. *The Design of Experiment*. New York: Hafner, 1935.

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

[7] 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.

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

[9] M. P. Heimdahl and G. Devaraj. Test-suite reduction for model based tests: Effects on test quality and implications for testing. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE)*, Linz, Austria, September 2004.

[10] A. Joshi and M. P. Heimdahl. Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier. In *SAFECOMP*, volume 3688 of *LNCS*, pages 122Ű–135. Springer-Verlag, Sept 2005.

[11] N. G. Leveson, M. Heimdahl, H. Hildreth, and J. Reese. TCAS II Requirements Specification.

[12] MathWorks. The MathWorks Inc. corporate web page. Via the world-wide-web: http://www.mathworks.com, 2004.

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

[14] S. Miller, A. Tribble, T. Carlson, and E. J. Danielson. Flight guidance system requirements specification. Technical Report CR-2003-212426, NASA, June 2003.

[15] S. P. Miller, E. A. Anderson, L. G. Wagner, M. W. Whalen, and M. P. Heimdahl. Formal verification of flight critical software. In *Proceedings of the AIAA Guidance, Navigation and Control Conference and Exhibit*, August 2005.

[16] S. P. Miller, M. P. Heimdahl, and A. Tribble. Proving the shalls. In *Proceedings of FM 2003: the 12th International FME Symposium*, September 2003.

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

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

[19] 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.

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

[21] *ARP 4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*. SAE International, December 1996.

[22] RTCA SC-205 (Joint with EUROCAE WG-71) Software Considerations. http://www.rtca.org/comm/Committee.cfm?id=55.

[23] M. Whalen. Autocoding tools interim report. In *NASA Contract NCC-01-001 Project Report*, February 2004.