Masking Boundary Value Coverage: Effectiveness and Efficiency

P. Vijay Suman[1], Tukaram Muske[1], Prasad Bokil[1], Ulka Shrotri[1], and R. Venkatesh[1]

TRDDC
54B, Hadapsar
Industrial Estate,
Pune 411 013, India

Abstract. Boundary value testing in the white-box setting tests relational expressions with boundary values. These relational expressions are often a part of larger conditional expressions or decisions. It is therefore important, for effective testing that the outcome of a relational expression independently influences the outcome of the expression or decision in which it is embedded. Extending MC/DC to boundary value testing was proposed in the literature as a technique to achieve this independence. Based on this idea, in this paper we formally define a new coverage criterion - masking boundary value coverage (MBVC). MBVC is an adaptation of masking of conditions to boundary value testing. Mutation based analysis is used to show that test data satisfying MBVC is more effective in detecting relational mutants than test data satisfying BVC.

In this paper, we give a formal argument justifying why test data for MBVC is more effective compared to that for BVC in detecting relational mutants. We performed an experiment to evaluate effectiveness and efficiency of MBVC test data relative to that for BVC, in detecting relational mutants. Firstly, mutation adequacy of the test set for MBVC was higher than that for BVC in 56% of cases, and never lower. Secondly, the test data for MBVC killed 80.7% of the total number of mutants generated, whereas the test data for BVC killed only 70.3% of them. A further refined analysis revealed that some mutants are such that they cannot be killed. We selected a small set of mutants randomly to get an estimate of percentage of such mutants. Then the extrapolated mutation adequacies were 92.75% and 80.8% respectively. We summarize the effect of masking on efficiency. Details of the experiment, tools developed for automation and analysis of the results are also provided in this paper.

1 Introduction

Bugs in software can be very expensive and can result in loss of life in the case of safety-critical software, product recalls in the case of embedded software and loss to business in other cases. Ensuring that a software application is bug-free is very difficult. Formal specification and verification of the entire software of such systems could be one way of demonstrating that an application is bug-free. However formal specification of the complete software is often unavailable, and in many cases not derivable. Moreover, when the program is complex, formally showing its correctness is often intractable.

Software testing serves as a practical and economic alternative to detect bugs in software. Testing methods have been extensively studied in the literature [13]. Testing typically assesses the end-to-end quality of a software by exercising the software on a representative set of test set to check if it meets the specified requirements. However, it is impossible to check program performance on all possible input data. An ideal test set should be large enough to effectively exercise most of the program executions, yet small enough so that the tester can comfortably run the program on it and compare the actual outputs with the expected outputs.

Generating effective test data efficiently is a challenging task. Test generation is usually done with respect to some coverage criterion. The coverage could be either functional coverage such as boundary value analysis (BVA) or structural coverage of the code. There are various types of structural coverage criteria [9] such as statement, condition, decision, condition/decision, modified condition/decision coverage (MC/DC), multiple condition coverage (MCC). Ensuring that a test set satisfies a coverage criterion is an indirect measure of its effectiveness [10].

Boundary value analysis is a functional testing technique in which tests include boundary values of relational expressions. A test set covers the boundary values for a relational expression `e relop c`, where `c` is a constant if `e` evaluates to `c`, `c+1` and `c-1` for some test in the set. A test set is said to achieve boundary value coverage (BVC) for a program if it covers every boundary value of every relational expression. BVC is useful for uncovering bugs in relational conditions where either an incorrect operator is used or there is an off-by-one error.

```
1. int inp1, inp2, op;
2. void func()
3. {
4.    if( ( inp1 > 10 ) && inp2 )
5.       op = 1;
6.    else
7.       op = 0;
8. }
```

Fig. 1. Running Example

Relational expressions in a program often occur as part of a larger conditional expression or decision. In such cases a test case that covers a boundary value of the expression should ensure that the expression has an independent effect on the outcome of the larger conditional expression or decision. Masking of conditions, as introduced in the context of masking MC/DC [9], can be applied to achieve this independent effect. A condition `c` in a decision `d` is said to be masked if the values of the other conditions in `d` are such that changing the value of `c` will have no impact on the result of `d`. The idea of extending MC/DC to relational testing was proposed in [4]. In this paper we formalize this idea by defining a new coverage criterion called masking boundary value

coverage (MBVC) that extends boundary value coverage with masking. In MBVC each test set that covers a relational expression for boundary values also ensures that the values of other conditions in the decision are such that the relational expression of interest is the one that influences the outcome of the decision. For example, consider the function `func` given in Figure 1. The relational condition of interest is `(inp1 > 10)` in line number 4. The difference between the two test sets for BVC and MBVC will be in the value of `inp2`. The test set for MBVC will ensure that the value of `inp2` is `non-zero` so that the relational expression has an independent effect, whereas the test set for BVC could have any value for `inp2` including `zero`.

The paper presents a more rigorous comparison of the effectiveness of test sets generated for BVC and MBVC using mutation analysis [2, 4, 7, 11]. We do this using a mutation operator that introduces off-by-one errors and relational operator errors in the code. We compare the two sets by their effectiveness in detecting erroneous versions, and how efficiently they can be generated. A formal argument is presented to show that where MBVC test data for a boundary value cannot be generated, the corresponding mutants cannot be killed by the BVC test data. Furthermore, when MBVC test data for a boundary value can be generated it ensures that the intermediate states in the corresponding mutants are different from that of the original, which is not necessarily true with BVC test data. It is still theoretically possible that test data for BVC would kill a mutant, but test data for MBVC would fail to kill the same mutant. However, this is not a side-effect of using masking, but rather an incidental possibility. For our experiments we chose a random set of 74 functions (which we refer to as *representative set*) from an embedded automotive application. Since we used a restricted mutation operator we could do an exhaustive mutation analysis as opposed to selecting a set of mutants randomly. This removes the repeatability requirement from our method of analysis.

This paper...

- formally defines boundary value coverage,
- introduces masking boundary value coverage,
- describes a white-box method to generate at unit level, test data satisfying MBVC,
- analyses the effectiveness and efficiency of masking in detecting relational mutants,
- describes the experimental setup and the tools used to perform the experiment, and
- presents the findings of the experiment along with a detailed analysis.

Related Work: Boundaries are common locations for errors that result in software faults and hence they need to be exercised by the test set. There have been studies comparing BVA with equivalence partitioning [15], and it is widely believed that BVA is very effective and necessary. Traditionally BVA is applied to the input specification to detect bugs in specification of input ranges. Automatic white-box generation of test data satisfying boundary value coverage (with or without masking) has never been attempted before. Masking of conditions has been used as a mechanism to achieve MC/DC coverage [1, 3], however, masking by itself is useful in generating more effective test data. Extending MC/DC to relational testing with boundary values was proposed in [4]. However, neither formal nor experimental analysis of effectiveness of masking in detecting was done as part of the paper. There are numerous works which use mutation analysis to experimentally compare the effectiveness and efficiency of various testing techniques.

See [2, 11, 7] for instance. The closest work to ours among these is that of [7]. However, our approach differs from these works in two ways. Firstly, we have not come across a mutation based comparision of coverage criteria like ours where the test data satisfying both the criteria are exhaustively generated using a white-box technique. This is different from more prevalent techniques of black-box generation where test data is generated randomly until a satisfactory level of required coverage is achieved. Secondly, we use a specific mutation operator which simulates only relational bugs. This is because we did not have access to appropriate mutation analysis tools at the point of conducting these experiments, in order to consider a more comprehensive set of mutation operators [16]. This reduces the total number of possible mutants, and enables us to give a formal argument why test set satisfying MBVC is more effective in detecting relational mutants. On the flip-side, this might not give an accurate comparision with respect to detecting mutants in general. However, we believe that MBVC test data would still turn out to be relatively more effective than BVC test data, if the experiments were repeated with a comprehensive set of mutation operators.

The primary contributions of our current work are the following. (1) We describe a white-box technique to generate test sets for both boundary value coverage and masking boundary value coverage. (2) We give a case wise analysis of scenarios wherein test data for MBVC kills more mutants than that for BVC. We also show a purely theoretical and incidental scenario where the vice-versa holds. (3) Our detailed experimental analysis demonstrated that MBVC test set is more effective in detecting relational mutants.

2 Masking Boundary Value Coverage

In this section we formally define boundary value coverage and masking boundary value coverage. We use the example function `func` in Figure 1 to illustrate the basic ideas. The input variables for `func` are `inp1` and `inp2`, whereas `op` is the output variable.

2.1 Boundary Value Coverage

A relational condition in the programming language `C` is an expression using a relational operator ($==, !=, <, >, <=, >=$) to compare two entities. To test whether a relational condition is coded correctly, the recommended approach is to generate test data which exercise its boundary situations. The hypothesis is that boundary test cases find (1) *off-by-one* errors and (2) *incorrect operator* errors. An off-by-one error is an instance where one of the operands is either one less or one more than the intended. Any instance where the intended relational operator was replaced by another operator, is referred to as an incorrect operator error. A *relational bug* is the presence of either an off-by-one error or an incorrect operator error. Relational testing is testing intended to find relational bugs.

Consider the condition `(inp1 > 10)` in line 4 of the function given in example code shown in Figure 1. The condition has an off-by-one error if the constant in the comparison should have been `9` or `11`. To detect this bug, test set should be generated such that `inp1` takes the values `10` and `11`, respectively. Similarly, for the instances of incorrect operator errors to be found, test set should be generated such that `inp1`

takes the values `9` and `10`. In other words, it is necessary to generate test set with `inp1` taking values `9`, `10` and `11` to detect all relational bugs.

Let r be a relational condition of the form $e_1 \sim e_2$ at program point p in a function $f(i_1 \ldots, i_n)$ where,

- $\sim$ is a relational operator and
- e_1 and e_2 are arbitrary integral expressions
- $i_1, \ldots, i_n$ are inputs to f

Definition 1 (BVC). *A test set TS satisfies BVC for r iff there exist three test vectors $t_1, t_2, t_3 \in TS$ such that e_1 evaluates to $e_2 - 1$, e_2 and $e_2 + 1$ at p for t_1, t_2 and t_3 respectively. Each t_i is a tuple of values $(v_1 \ldots v_n)$ for the input variables $i_1 \ldots i_n$.*

A test set that satisfies BVC for each relational condition in a function is said to provide *boundary value coverage* for that function. This definition can be similarly extended for a program.

It is noteworthy that the boundary value test set would not be sufficient to detect instances where both kinds of errors can occur simultaneously. For instance, the conditions `(inp1 > 10)` and `(inp1 == 11)`, have uniform truth value for the three test set entries described above.

2.2 Masking Boundary Value Coverage

When a relational expression occurs as part of a larger conditional expression, BVC may not be sufficient to detect relational bugs. To address this problem we introduce masking boundary value coverage, which adopts the idea of masking from masking MC/DC to BVC. Let d be a decision with several conditions in it and let r be one condition in it. If the test set generated is such that the value of r does not have an independent effect over the outcome of d, then the effect of r is not seen on the output either. To ensure this independent effect, other conditions in d should be masked while generating test set for r.

Let $r = e_1 \sim e_2$ be an atomic relational condition in a decision d at program point p in a function $f(i_1 \ldots i_n)$. Let $d^{r'}$ represent the decision generated by replacing the condition r by r', and let d_t represent the truth value of d at program point p with the test vector t.

Definition 2 (MBVC). *A test set TS satisfies MBVC for r iff there exist three test vectors $t_1, t_2, t_3 \in TS$,*

1. *e_1 evaluates to $e_2 - 1$, e_2 and $e_2 + 1$ respectively at p, for t_1, t_2 and t_3 respectively. Each t_i is a tuple of values $(v_1 \ldots v_n)$ for the input variables $i_1 \ldots i_n$ and*
2. *for each t_i, $d^r_{t_i} = \neg d^{\neg r}_{t_i}$.*

A test set that satisfies MBVC for each relational condition in a function is said to provide *masking boundary value coverage* for that function. This definition can be similarly extended for a program.

For example, consider the decision in line 4 in the afore-mentioned code. To ensure BVC for the condition `(inp1 > 10)`, the test set should be such that this condition

is exercised with `inp1` taking the values `9`, `10` and `11`. At the same time, to see the effect of a coding error on the output, it is necessary that `inp2` takes a non-zero value in this test set.

Claim: Test data generated for MBVC is more effective at detecting relational mutants than that generated for BVC.

3 Mutation Analysis

Observability based techniques such as mutation analysis are most suitable to compare different coverage criteria. Mutation analysis has been used in the past to determine test suite adequacy [12, 11] and also to compare test coverage criteria [2, 7].

Mutation [12] is a way of purposely modifying code of a function so as to change its external behaviour. The modified function is called a mutant. Mutation operators define the way in which a particular programming construct gets modified. For instance, a mutation operator can be defined which replaces the operator in a randomly selected relational condition with a different one. Depending on which program unit is selected and how it is mutated different mutants of the same function can be created. For instance, if there are n relational operators in a function, there would be $5n$ possible mutants of this function with the above mutation operator. Note that we work under the assumption that a mutant contains exactly one fault [12, 6].

A test set is said to *kill* a mutant iff there exists a test data entry in the test set such that

- Both the original program and the mutant terminate on the test data, and
- The output of the mutant is different from the output of the original program when run with this test data.

Note that a test data entry kills a mutant only if both the following conditions hold [12].

1. It causes different program states for the mutant and the original program (*Weak Mutation* [8]).
2. This difference in state propagates to the output of the program.

The number of mutants killed by a test set is a measure of effectiveness of the test set. Mutation adequacy ratio represents this effectiveness. Formally -

Definition 3 (Mutation Adequacy Ratio). *Let TS represent a test set, and let td represent a test set input. A mutant f' of the program f is said to be killed by a test set TS iff $\exists td \in TS$ such that the output of f and f' are not equal when run with the input td. The mutation adequacy ratio of a test set TS, $\mathcal{AM}_{\mathcal{F}}(TS)$, is the fraction of mutants killed by it in a given set of mutants $\mathcal{F}$.*

The code shown in Figure 2 gives one possible mutant of our running example. The constant used in the relational condition is perturbed by one.

BVC and MBVC are both coverage criteria intended to find relational bugs and hence for our experiments we use a mutation operator that only changes operators and

```
1. int inp1, inp2, op;
2. void func()
3. {
4.    if( ( inp1 > 9 ) && inp2 )
5.       op = 1;
6.    else
7.       op = 0;
8. }
```

Fig. 2. An Example Mutant

operands in relational expressions. Using such a restricted mutation operator as opposed to using a comprehensive set of mutation operators helps in greatly reducing the number of possible mutants to only those that are of direct interest to the coverage criteria under consideration.

Formally given a relational condition $e_1 \sim e_2$ the mutation operator is defined as

$$\mu(e_1 \sim e_2) = \begin{cases} e_1 \sim' e_2 \text{ where } \sim' \neq \sim \text{, or} \\ e_1 \sim e_2' \text{ where } e_2' \text{ is either } e_2 - 1 \text{ or} \\ \qquad e_2 + 1 \end{cases}$$

4 Generation of Test Sets and Mutants

This section describes the tools used for automatic test set generation and mutant generation.

4.1 Test Set Generation:

We have extended our tool AutoGen [3], to accept additional criteria BVC and MBVC and automatically generate test set for that criterion thus reducing the time and effort required to achieve coverage. AutoGen also attempts to generate a non-redundant test set by continuously performing an analysis of coverage that has already been achieved by the test set generated at each step and generating additional test data only for uncovered coverage units.

AutoGen transforms the source program to include statements that generate non-deterministic values for input variables and adds assert statements, the violation of which will result in a condition getting covered. To generate test set that violates these asserts the transformed program is analyzed using `C` Bounded Model Checker (CBMC) [5]. Given a `C` program and an assert statement, CBMC checks if the assert is valid and generates a trace that violates the assert if it is not valid.

CBMC is a SAT-based bounded model checker for programs written in Ansi-C and C++. Note that since property checking is undecidable in general, for some cases CBMC may not terminate or may run out of resources. For our experiments we considered only code for which CBMC terminates successfully for both BVC and MBVC.

The functions for which CBMC did not terminate were mostly ones with loops. Generating test data for these functions is necessary to keep our analysis fair. However, doing this would require techniques which are out of scope of this paper.

Code transformation: Source code transformation is explained using the example function `func` of Figure 1. A transformed code is shown in Figure 3. The code has been modified for generating values for input variables and for inserting assert statements. Please refer to [3] for detailed description of the transformation. The tool first identifies the set of input variables needed in the function and then the code is modified such that input variables take non-deterministic values at the start of execution of the function. Any function whose name has the prefix "`nondet_`" is treated as a special function by CBMC. Let `type` be the data type of the return value of such a function. CBMC assumes that the return value of this particular function can be any valid value in the range defined by `type`. Hence lines 5 and 6 of the instrumented code in Figure 3 essentially communicate to CBMC that it has to check executions of `func`, wherein `inp1` and `inp2` can both be assigned any two independent values from the data type `int`.

The input code is also annotated with `assert` statements which are of the form `assert(expr)`, where `expr` is any expression allowed by the Ansi-C syntax. To model check this assertion, CBMC tries to check whether there exists an assignment to the input variables, which causes the execution to satisfy `(expr==0)` when the corresponding line is reached. If so, the assertion is said to be violated, and CBMC reports the same along with a counter-example trace. The required test set is nothing but the values assigned to the input variables in the trace.

Asserts for BVC: Line 7 in Figure 3 shows the `assert` annotation. When the annotated `C` code in Figure 3 is given as input to CBMC, it produces a counter-example trace with `11` as the value for `inp1` and `0` as the value for `inp2`.

Asserts for MBVC: The test set for BVC is generated without considering the fact that `inp2` is part of the same decision (see line 4 in Figure 1). As explained in Section

```
1. int inp1, inp2, op;
2. int nondet_int();
3. void func()
4. {
5.     inp1 = nondet_int();
6.     inp2 = nondet_int();
7.     assert (!((inp1)==(10+1)));
8.     if( ( inp1 > 10 ) && inp2 )
9.        op = 1;
10.    else
11.       op = 0;
12. }
```

Fig. 3. Annotated Code for $e_1 \sim e_2$

2.2, one can use masking of conditions to overcome this limitation. To generate test set for MBVC, we need to ensure that `inp2` takes a non-zero value in the entire test set. In order to achieve this, we instrument the code further such that the `assert` statement is guarded by an appropriate `if` statement. The condition corresponding to this `if` statement is such that when the test set is generated, the effect of other conditions is masked. In case of our running example, line 7 in Figure 3, should be replaced by the following two lines.

```
if(inp2)
   assert (!((inp1)==(10+1)));
```

When the annotated `C` code with above mentioned modification is given as input to CBMC, it produces a counter-example trace with `11` as the value for `inp1` and `1` as the value for `inp2`.

4.2 Mutant Generation:

To generate mutants we have developed a tool that implements the mutation operator μ. The tool takes `C` source code as input and the mutant is generated by applying the mutation operator μ to each relational condition in the function. The tool also generates the `main` function which executes the mutated function on a given test set, and records the values of the output variables after each invocation.

In general, the number of possible mutants of a function can be quite large. However, we take the *do fewer* approach suggested in [12], by restricting to a particular mutation operator. The number of possible mutants of a function with n relational operators is only $7n$ using our mutation operator. Hence we generate the mutants of each function exhaustively for our analysis. It is noteworthy that the mutation operator emulates the kind of erroneous functions intended to be detected by the test sets generated for both BVC and MBVC.

4.3 Test Sets

The original code (Figure 1) of the actual function and the mutant code (Figure 2) is executed with the generated test set. Table 1 (on the left) shows the test set generated for BVC and it also shows the output of actual function and the mutant code. In this case it is an accident that our test data generator generated `0` for `inp2`.

Table 1 (on the right) shows the test set generated for MBVC and it also shows the output of actual function and the mutant code. It is easy to see that the mutant of Figure 2 gets killed by test set for MBVC but not by test set for BVC.

5 Formal Analysis of the Effect of Masking

This section presents a formal analysis of how masking of conditions helps in achieving additional effectiveness in detecting relational mutants. We consider an arbitrary mutant and do a case-wise analysis on the basis of generatability of each type of test data.

Let f be the function of interest, and let f' be one of its mutants. Let r be the relational condition $e_1 \sim e_2$ in decision d, which was changed by μ and let r' represent the variant of r and d' the modified decision. Let b represent a boundary value which differentiates r from r', and let τ_b and t_b respectively represent any MBVC and BVC test data for coverage of b.

Case 1 Neither τ_b nor t_b exist. The mutant cannot be detected by either of the test sets.

Case 2 τ_b exists, but not t_b. This contradicts the definition of MBVC, as any test data entry that satisfies MBVC, by definition satisfies BVC.

Case 3 t_b exists, but not τ_b. If t_b kills the mutant then since f and f' differ only in r, $r = \neg r'$ and $d = \neg d'$ must be true. Then t_b also satisfies MBVC for r which contradicts our assumption. Hence t_b cannot kill the mutant.

Case 4 Both τ_b and t_b exist.

(a) t_b kills f' and τ_b does not. We give an example of such a function and a mutant in Figure 4 to argue that this is a valid possibility. The commented code on line 6 represents one possible application of μ to `( inp1 > 10 )`. For this example $b = 10$. Table 2 shows two possible test data entries for this boundary value, one satisfying MBVC and the other satisfying BVC. This shows one possible instance where the mutant is killed by t_b but not by τ_b. Note that the same example can be used to show how τ_b kills a mutant that t_b does not.
(b) All the other sub cases do not contradict our claim and hence we do not analyze them.

All the cases are in favour of MBVC, except Case 4(a). Note that Case 4(a) is an instance where the test data for MBVC could not ensure the second condition for mutant to be detected (see Section 3). However, since `inp2` is a condition independent of the mutated condition, it is highly improbable that `inp2` has different values in the two test data entries, especially when test data is generated automatically using a fixed algorithm. Furthermore, if the value of `inp2` was same in the two test data entries,

inp1	**inp2**	**Function Output**	**Mutant Output**
9	0	0	0
10	0	0	0
11	0	0	0

inp1	**inp2**	**Function Output**	**Mutant Output**
9	1	0	0
10	1	0	1
11	1	1	1

Table 1. Test Data for BVC and MBVC

```
1. int inp1, inp2, op;
2. void func()
3. {
4. int local;
5.    if( inp1 > 10 )
6.    //if( inp1 > 9 )
7.       local = 1;
8.    else
9.       local = 0;
10.   if( local || inp2 )
11.      op = 1;
12.   else
13.      op = 0;
14. }
```

Fig. 4. Counter-Example

Coverage	`inp1`	`inp2`	Function Output	Mutant Output
MBVC	10	1	1	1
BVC	10	0	0	1

Table 2. Test Data

it cannot happen that BVC test data detects the mutant but MBVC test data does not. This argument is supported by our experimental data which contains no occurrences of scenarios similar to Case 4(a).

6 Experiment

To validate our theory an experiment based on mutation analysis was conducted. In this section we describe our experimental setup, and then present the empirical data supporting our claim. Section 6.2 gives the figures obtained in our experiments.

6.1 The Setup

To make our experiment more realistic we chose some modules of an embedded application. The application is a `C` program intended to control the battery in a car. The application has around 22 modules of which we randomly chose 12 for our analysis. Among the functions in these modules the number of functions which had logical combinations of relational conditions, and were amenable to our analysis was 74. The total number of lines of code for these 74 functions is 3646.

Let $\mathcal{F}$ represent the set of 74 functions we analyzed. The following steps were executed on each function f in $\mathcal{F}$.

1. Identify the set of output variables of f.
2. Generate test sets for f satisfying BVC and MBVC.
3. Generate a random subset of all possible mutants of f.
4. Compile and execute each of the mutants on both the test sets, and record independently the number of mutants killed by each of them. In our setup, since the test sets are executed on the same set of mutants, we can consider the number of mutants killed as a direct measure of their mutation adequacy ratios.

6.2 Empirical Data

Table 3 summarizes the empirical data collected from our mutation analysis. A description of the contents of various rows in the table is as follows:

- The first row gives the total number of functions in our representative set. Note that only functions involving decisions with logical combination of relational conditions were considered.
- The second row gives the number of functions for which MBVC killed more mutants than BVC.
- The third row gives the number of functions for which BVC killed more mutants.
- The fourth row gives the number of functions for which both BVC and MBVC killed the same number of mutants.
- The fifth row gives the total number of mutants generated using our mutation operator.
- The sixth row gives the number of mutants killed by test data satisfying MBVC.
- The seventh row gives the number of mutants killed by test data satisfying BVC.

The rest of the rows are explained below.

We did a further (manual) investigation of why 893 of these mutants were not killed by any of the generated test data. We had the time to analyze 153 of them, and it was revealed that 103 of these were not killed because the test data that can kill these mutants cannot be generated. For instance, the actual relational condition in a function was of the form `b==1`, whereas that in the mutant was `b>=1`. The boundary value that can kill this mutant is `2`. But the variable `b` was a bit field of size `1`. Extrapolating these numbers the total number of mutants that can be killed comes down to 4026. And hence the mutation adequacies of test data satisfying MBVC and BVC are 92.75% and 80.8% respectively. The rows 8,9 and 10 in Table 3 quote these numbers.

Propagation The reason for the other mutants not getting killed by test set satisfying MBVC was that the change in state induced by the test data did not propagate to the output. This was because of masking by a condition in an independent decision. For instance consider the code given in Figure 5. The commented line 6 is a mutated version of the code in line 6. The boundary value for `inp1` that can detect this mutant is `11`. However, if the test values that were generated for `inp1`, `inp2` and `inp3` were `11`, `1` and `1`, the output of the mutated function does not differ from that of the original. Note that the change in state is observed at line 5. But this fails to propagate to the output due to the input variable `inp3` taking the value `1`. It is noteworthy that the condition `inp3` is part of an entirely different decision.

Efficiency Using masking for relational testing comes with the following added costs. For the functions wherein the conditions use logical operators,

– the **size** of the test set for MBVC was 28.65% more than that for BVC. This happens mostly when there are more than one relational expression in a decision. In such cases the probability of a test vector generated for covering one component of a decision covering some other component also goes down in presence of masking. Hence more test vectors are required to provide MBVC coverage.

Table 3. Empirical Data from Mutation Analysis

No.	Category	Number	Percentage
1	Representative Set	74	-NA-
2	$\mathcal{AM}(TS_M) > \mathcal{AM}(TS)$	42	***56***
3	$\mathcal{AM}(TS_M) < \mathcal{AM}(TS)$	0	***0***
4	$\mathcal{AM}(TS_M) = \mathcal{AM}(TS)$	32	44
5	Total Mutants	4627	-NA-
6	Mutants killed by MBVC test data	3734	80.7
7	Mutants killed by BVC test data	3253	70.3
8	Number of Detectable Mutants (Extrapolated)	4026	-NA-
9	Mutants killed by MBVC test data	3734	92.75
10	Mutants killed by BVC test data	3253	80.8

– the **time** taken to generate the test set for MBVC was 9.38% more than that taken to generate for BVC. The average time taken for generating test data for MBVC was 95.19 seconds per function, as opposed to 86.26 seconds per function for BVC. This was expected because the constraint that is generated by the model checker will be stronger when masking is used. Computing a satisfiable assignment for this constraint is expected to be relatively expensive.

However, these costs should be acceptable as the probability of detecting a bug goes up considerably and although the time taken for MBVC is more, the entire generation is automatic.

In the industrial code we tried to target, there were functions which could not be analyzed using our approach. For example, there were instances where the model checker ran out of resources. We completely excluded data corresponding to any such function.

7 Conclusions and Discussions

Mutation analysis validated the claim that MBVC is more effective than simple BVC in detecting mutants. We restricted our attention to the test set generated for BVC of relational conditions occurring in the code. Masking is likely to be beneficial in other contexts too such as the data flow coverage criteria p-use [14].

Our experiments, modulo the limitations (see Section 7.1), suggest that using masking when generating boundary value test data is more effective and at the same time insignificantly inefficient. Following are some of our observations.

– Test data for MBVC is strictly more effective in 56% of the cases.
– The average mutation adequacies of MBVC test data and BVC test data are 80.7 and 70.3 respectively. A further investigation revealed that there were mutants

```
1. int inp1, inp2, op;
2. void func()
3. {
4. int local;
5.    if( inp1 > 10 && inp2)
6.    //if( inp1 > 11 && inp2)
7.       local = 1;
8.    else
9.       local = 0;
10.   if( local || inp3 )
11.      op = 1;
12.   else
13.      op = 0;
14. }
```

Fig. 5. Hindrance in Propagation

that could not be killed, because test data for killing these cannot be generated. Discounting these, with some extrapolation, the mutation adequacies turned out to be 92.75 and 80.8 respectively.
- The average size of the test data generated increased by 28.65% and the average time taken to generate the test data increased by 9.38%. Given the increase in chances of detecting a bug, these added prices are acceptable. Moreover, the test data generation was automated in the form of a tool and hence required insignificant manual effort.
- The defects that could not be detected by MBVC test data were diagnosed to be those in which the change in state induced by the test data was hindered from propagating to the output by an independent condition occurring in a decision other than the one of interest.

We would like to point out that when none of the conditions in the code is such that it is a logical combination of atomic conditions, masking has neither positive nor adverse effects.

7.1 Limitations of our Approach

- One prominent limitation of our approach is that we consider only those functions for which the model checker terminated. A more fair analysis would require significant efforts towards loop abstraction and is out of scope of this paper.
- Another factor which could have improved the credibility of our analysis is keeping the size of the test sets being compared same. This could be achieved by padding more test data to the BVC test set to make it as big as the test set for MBVC.

7.2 Future Directions

- Other coverage criteria such as p-use can be extended with masking of conditions. We intend to do more rigorous experimental study of how effective masking would be in those cases.
- Using specific mutation operators to assess the effectiveness of a test suite in finding the bugs it is intended to find, is an interesting idea by itself. When the category of bugs to be detected is clearly defined, this approach is a useful alternative to using a comprehensive set of mutation operators. A systematic study of how effective our alternative approach is in the spirit of [16] needs to be done.
- We would like to point out that the test data generation is at unit level and does not take procedure preconditions into account.

References

1. Do-178b: Software considerations in airborne systems and equipment certification, 1982.
2. James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Trans. Softw. Eng.*, 32(8):608–624, 2006.

3. Prasad Bokil, Priyanka Darke, Ulka Shrotri, and R. Venkatesh. Automatic test data generation for c programs. *Secure System Integration and Reliability Improvement*, 0:359–368, 2009.
4. John Joseph 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, 2001.
5. Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.
6. Richard A. DeMillo, Richard J. Lipton, and Frederick Gerald Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.
7. Atul Gupta and Pankaj Jalote. An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing. *Int. J. Softw. Tools Technol. Transf.*, 10(2):145–160, 2008.
8. W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Trans. Softw. Eng.*, 8(4):371–379, 1982.
9. Hayhurst Kelly J., Veerhusen Dan S., Chilenski John J., and Rierson Leanna K. A practical tutorial on modified condition/decision coverage. Technical Report NAS 1.15:210876, 2001.
10. Glenford J. Myers. *The Art of Software Testing*. Wiley, New York, 1979.
11. Akbar Siami Namin and James H. Andrews. The influence of size and coverage on test suite effectiveness. In *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 57–68, New York, NY, USA, 2009. ACM.
12. A. Jefferson Offutt and Ronald H. Untch. Mutation 2000: uniting the orthogonal. pages 34–44, 2001.
13. William Perry. *Effective methods for software testing*. Wiley-QED Publishing, Somerset, NJ, USA, 1995.
14. Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, 11(4):367–375, 1985.
15. Stuart C. Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In *METRICS '97: Proceedings of the 4th International Symposium on Software Metrics*, page 64, Washington, DC, USA, 1997. IEEE Computer Society.
16. Akbar Siami Namin, James H. Andrews, and Duncan J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 351–360, New York, NY, USA, 2008. ACM.