Mutation Operators for Specifications

Paul E. Black
National Institute of
Standards and Technology
Gaithersburg, MD 20899
paul.black@nist.gov

Vadim Okun Yaacov Yesha
University of Maryland
Baltimore County
Baltimore, MD 21250
{vokun1,yayesha}@cs.umbc.edu

Abstract

Testing has a vital support role in the software engineering process, but developing tests often takes significant resources. A formal specification is a repository of knowledge about a system, and a recent method uses such specifications to automatically generate complete test suites via mutation analysis.

We define an extensive set of mutation operators for use with this method. We report the results of our theoretical and experimental investigation of the relationships between the classes of faults detected by the various operators. Finally, we recommend sets of mutation operators which yield good test coverage at a reduced cost compared to using all proposed operators.

1 Introduction

A formal specification is a repository of knowledge about a system. In particular, a specification provides valuable information for testing programs. For instance, specification-based testing may detect a missing path error [12], that is, a situation when an implementation neglects an aspect of a problem, and a section of code is altogether absent. Since there is no evidence in the code itself for the omission, such errors are very hard to find by analyzing the code alone. Further, code-based testing is not possible for some systems because testers do not have access to the source code. Additionally, generating tests from a specification can proceed independently of program development, and the created tests apply to all implementations of the specification, e.g., ports.

Ammann and Black described a novel method using a combination of model checking and mutation analysis to automatically produce tests from formal specifications [2] and measure test coverage [1]. The test cases considered in the method constitute a complete test suite, that is, all test cases include both inputs and expected results. Model checking is a formal technique for verifying that temporal logic expressions are consistent with all executions of a state machine. Mutation analysis [9] uses mutation operators to introduce small changes, or mutations, into the specification, producing mutant specifications. Better test sets are those which reveal more mutants.

Ammann and Black defined a few mutation operators for formal specifications, but did not consider their relative merits. In this paper, we describe a larger set of mutation operators, including some new ones. We compare, both theoretically and empirically, the effectiveness of the operators and the number of mutations they produce. For the theoretical comparison, we extend Kuhn's analysis of fault classes [14] and tie it to mutation operators.

1.1 Software Testing and Model Checking

Model checking is a formal verification technique based on state exploration. A model checking specification consists of two parts. One part is a state machine defined in terms of variables, initial values for the variables, environmental assumptions, and a description of the conditions under which variables may change value. The other part is temporal logic expressions over states and execution paths. Conceptually, a model checker visits all reachable states and verifies that the temporal logic expressions are satisfied over all paths. When an expression is not satisfied, the model checker generates a counterexample in the form of a trace or sequence of states, if possible.

Although model checking began as a method for verifying hardware designs, there is growing evidence that it can be applied to specifications for large software systems, such as TCAS II [7]. In addition to verifying properties of software, model checking is being applied to test generation and test coverage evaluation [2, 6, 10, 11].

In both uses, one begins with selection of a test criterion [12], that is, a decision about what properties of a specification must be exercised to constitute a thorough test. Some specification-based test criteria are conjunctive com-

plementary closure partitions [6], branch coverage [11], and mutation adequacy [1].

The chosen test criterion is applied to the specification to derive test requirements, i.e., a set of individual properties to be tested. To use a model checker, these requirements are represented as temporal logic formulas. To evaluate coverage of a test set, each test is turned into an execution sequence, and the model checker determines which requirements are satisfied by the execution. See [1] for details.

To generate tests, the test criterion is applied to ultimately yield negative requirements, that is, requirements which are considered satisfied if they are inconsistent with the state machine. When the model checker finds a requirement to be inconsistent, it produces a counterexample if possible. The counterexamples contain both stimulus and expected values, so they may be automatically converted to complete test cases.

Specification-Based Mutation Adequacy

Mutation adequacy is a test criterion which naturally yields negative requirements. First, a set of temporal logic expressions restating, or reflecting, the state machine's transition relation is derived mechanically [1]. This set, together with pre-existing expressions, if any, is consistent with the state machine and comprises the part of the specification to be mutated.

A mutant specification is produced by applying a single mutation operator once to the temporal logic portion of the specification. Applying operators repeatedly yields a set of mutants. The mutants represent negative requirements, so they can be used for both test generation and evaluation.

The SMV Model Checker

We chose a popular model checker, SMV [15]. It uses Computation Tree Logic (CTL) [8], which is a branching-time temporal logic extending propositional logic with temporal operators.

Here is a short SMV example. "Request" is an input variable, and "state" is a scalar with possible values "ready" and "busy." The initial value of state is "ready." The next state is "busy" if the state is "ready" and there is a request. Otherwise the next state is "ready" or "busy" nondeterministically. The SPEC clause is a CTL formula which states that whenever there is a request, state will eventually become "busy."

```
MODULE main
VAR
    request : boolean;
    state : {ready, busy};
ASSIGN
    init(state) := ready;
    next(state) := case
        state = ready & request : busy;
        1 : {ready, busy};
    esac;
SPEC AG (request -> AF state = busy)
```

The choice of model checker forces the specification to be in the language which the model checker accepts. Some might object that SMV's state machine description is at too low a level for practitioners to use, and we agree. A practical system must extract state machines from higher level descriptions such as SCR specifications [3], MATLAB stateflows [4], or UML state diagrams.

1.2 Hierarchy of Fault Classes

Mutation operators are related to the set of fault classes which Kuhn analyzed [14]. The set includes:

- Variable Reference Fault (VRF) - replace a Boolean variable x by another variable y, $x \neq y$.
- Variable Negation Fault (VNF) - replace a Boolean variable x by $\bar{x}$.
- Expression Negation Fault (ENF) - replace a Boolean expression p by $\bar{p}$.
- Missing Condition Fault (MCF) - a failure to check preconditions.

Earlier Kuhn developed a technique [13] based on predicate difference for analyzing effects of changes in formal specifications. Applying this technique, he derived a hierarchy of fault classes used in specification-based software testing.

The detection conditions for a predicate P are the conditions under which a change to P affects the value of the predicate P. A test detects an error if and only if a faulty predicate P' evaluates to a different value than the correct predicate P, e.g., $P \oplus P'$.

Let P_e^x be a predicate P with all free occurrences of variable x replaced by expression e. Let F be a fault in which x is replaced by e. The fault F is detected under the condition $P \oplus P_e^x$.

If S is a specification in disjunctive normal form (DNF), the conditions for detecting VRF, VNF, and ENF are:

- $S_{VRF} = S \oplus S_y^x$, where x and y are distinct variables in S, and y is substituted for x.
- $S_{VNF} = S \oplus S_{\bar{x}}^x$, where x is a variable in S.
- $S_{ENF} = S \oplus S_{\bar{X}}^X$, where X is an expression in S.

The relationships between detection conditions are:

If the variable replaced in S_{VRF} is the same variable negated in S_{VNF}, then $S_{VRF} \rightarrow S_{VNF}$.

If all expressions containing the variable negated in S_{VNF} are negated in S_{ENF}, then $S_{VNF} \rightarrow S_{ENF}$.

Kuhn concludes that any test that detects a VRF for a variable in a predicate also detects a VNF for the same variable, and any test that detects a VNF for a variable also detects an ENF for the expression in which the variable occurs.

We use Kuhn's approaches and theoretical conclusions to analyze mutation operators. Section 2 defines mutation operators for specifications together with their respective fault classes. Section 3 investigates the relationships between detection conditions for several fault classes analytically and compares the effectiveness of the mutation operators experimentally. We present our conclusions about the relative merit of mutation operators in Section 4.

2 Specification Mutation Operators

We use the following overall guiding principles [17] to formulate and implement our mutation operators:

1. Mutation categories should model potential faults.
2. Only simple, first order mutants should be generated.
3. Only syntactically correct mutants should be generated.
4. The user should have control over the selection of which mutation categories to apply at any one time.

The first principle means it is important to recognize different types of faults. While presenting mutation operators, we state which specification fault classes are modeled by the operators. In fact, each mutation operator is designed to uncover faults belonging to the corresponding fault class.

Since we are interested in relating our work to the theoretical results obtained in [14], we define the mutation operators so that their respective fault classes closely correspond to those definitions. For example, consider the following fault. The constant c_1 is replaced with constant c_2 in the expression $x = c_1$, where x is a variable. If Boolean variables a_1 and a_2 represent $x = c_1$ and $x = c_2$, respectively, then this is a variable reference fault (VRF). To account for this and similar cases, we need the definitions below.

2.1 Definitions

We define, similarly to [16], a simple expression as one of the following, possibly negated:

A Boolean variable.

An expression $token1 = token2$, where $token1$ and $token2$ are either a variable of type scalar or a constant, e.g., $state = busy$, where $state$ is a variable and $busy$ is a constant from the domain of $state$.

A simple relational expression: $token1\ operator\ token2$, where $token1$ and $token2$ are either a variable of type integer or a constant, $operator$ is one of $<, \leq, =, =, >$, or $\geq$.

A compound expression consists of at least one binary Boolean operator (including conjunction, disjunction) connecting two or more expressions, and possibly negation operators and parentheses.

We consider two kinds of operands in CTL: state variables and symbolic constants. State variables may be of Boolean, scalar or integer type. Value of a scalar variable is drawn from a finite set of constants. An integer variable takes value from a range. An SMV specification may also contain symbolic constants defined by the user to represent integers.

We define UT_{OPER} to be the set of unique traces generated by mutation operator $OPER$.

Additionally, the following notation is used throughout the paper:

$\vee$ and $\wedge$ represent disjunction and conjunction respectively in the formulas. However, when presenting SMV specifications we use instead $|$ and &, since they are a part of SMV syntax.

$\rightarrow$ represents implication, $\oplus$ represents exclusive or.

1 and 0 are used to denote "true" and "false," respectively.

2.2 Categories of Mutation Operators

Each fault class has a corresponding mutation operator. Applying a mutation operator gives rise to a fault in that class. For example, instances of the missing condition fault (MCF) class can be generated by a missing condition operator (MCO). Note that the abbreviation of the mutation operator ends in O, and the abbreviation of the corresponding fault class ends in F. Below we define mutation operators for common fault classes.

Although mutation operators are independent of any particular specification notation, here we present them for CTL specifications. Illustrative mutants for each operator are shown in Table 1.

Operand Replacement Operator (ORO).

Replace an operand, that is, a variable or constant, by another syntactically legal operand.

Do not replace the operand if it results in a constant ($c1$ $operator$ $c2$) or reflexive (x $operator$ x) expression, since an equivalent mutant is produced by applying the Stuck-At operator described below. Do not replace a number with another number, since this may result in too many mutants.

Simple Expression Negation Operator (SNO).

Replace a simple expression by its negation.

Expression Negation Operator (ENO).

Replace an expression by its negation. Temporal expressions, such as AG and EF, are not negated since SMV does not produce counterexamples from such mutants.

Logical Operator Replacement (LRO).

Replace a logical operator ($\&, |, \rightarrow$) by another logical operator.

Relational Operator Replacement (RRO).

Replace a relational operator ($<, \leq, >, \geq, =, =$) by any other relational operator, except its opposite. For example, do not replace $<$ with its opposite, $\geq$, because that is the same as negating the expression. Only replace $=$ or $=$ when applied to an integer expressions.

Missing Condition Operator (MCO).

Delete conditions (only simple expressions) from conjunctions, disjunctions, and implications.

Stuck-At Operator (STO).

This consists of two operators: stuck-at-0, replace a simple expression with 0, and stuck-at-1, replace a simple expression with 1.

Associative Shift Operator (ASO).

Change the association between variables, e.g., replace $x \rightarrow y_1 \& y \ \& y_3$ with $(x \rightarrow y\ ) \& y_2 \& y_3$. We do not replace the formula with $(x \rightarrow y\ \& y\ ) \& y_3$. This reduces the number of mutants generated by ASO.

Table 1 contains mutants generated from three formulas: the CTL formula presented in Section 1.1, the formula "AG (x & y $\rightarrow$ z)" (by ASO), and the formula "AG (WaterPres $<$ 100)" (by RRO).

If the number of atoms (variables and constants) in a specification is V and the number of value references is , ORO results in $O(V *\ )$ mutants, whereas SNO, LRO, MCO, STO, ASO and RRO result in $O(\ )$ mutants.

Operator	Example Mutants
ORO	AG (request $\rightarrow$ AF state = ready)
SNO	AG (!request $\rightarrow$ AF state = busy) AG (request $\rightarrow$ AF (!state = busy))
ENO	AG (!(request $\rightarrow$ AF state = busy))
LRO	AG (request & AF state = busy) AG (request | AF state = busy)
MCO	AG AF state = busy
STA	AG (0 $\rightarrow$ AF state = busy) AG (1 $\rightarrow$ AF state = busy) AG (request $\rightarrow$ AF 0) AG (request $\rightarrow$ AF 1)
ASO	AG (x & (y $\rightarrow$ z))
RRO	AG (WaterPres $<=$ 100) AG (WaterPres 100) AG (WaterPres $=$ 100) AG (WaterPres ! $=$ 100)

Table 1. Mutation Operators and their Illustrative Mutants.

2.3 Correspondence to Kuhn's Fault Classes

Our mutation operators generally do not correspond exactly to Kuhn's fault classes [14]. Consider a fault when expression $x < c$ is replaced with $x \quad c$, where x is a variable and c is a constant. If we have boolean variables represent $x < c$ and $x \quad c$, this is a variable reference fault (VRF).

If we combine ORO and RRO into a single operator, ORO$^+$, this new operator generates a class of faults closely matching VRF. We call its corresponding fault class ORF$^+$.

For analysis, we define a mutation operator which generates a class of faults identical to VRF. This Simple Expression Replacement Operator (SRO) replaces a simple expression by every other syntactically valid simple expression of atoms in the model.

SRO sometimes generates higher order mutants, so by Woodward's principle [17], it should not be used for test generation. Additionally, the operator produces a very large number of mutants. Not surprisingly, SRO generates a set of mutants which includes those of ORO$^+$, thus $UT_{ORO^+} \subseteq T_{SRO}$.

3 Comparison of Mutation Operators

In this section we analyze the relationships between several fault classes for restricted form of specifications, and we study the mutation operators experimentally.

3.1 Theoretical Comparison of Fault Classes

Analysis of Faults in Formulas

For analysis, we only consider specifications with formulas in disjunctive normal form (DNF), i.e., $a\ a\ .\ .a_k\ \ b\ b\ .\ .b_m\ \ \cdots\ \ z\ z\ .\ \ z_n$, where $a_i, b_j, .\ ., z_l$ are simple expressions. Here and in the theorem proof below, $\wedge$ is sometimes omitted, e.g., $a\ a\ .\ .a_k$ is a shorthand for $a\ \wedge a\ \wedge .\ .\wedge a_k$.

Let x be a simple expression in a formula S, and X be a possibly compound expression in S. Here are the detection conditions for several fault classes:

$S_{SNF} = S \oplus S^x_{\bar{x}}$.

$S_{ENF} = S \oplus S$ - .

$S_{STF} = S_{ST0} \wedge S_{ST}$, where $S_{ST0} = S \oplus S^x_0$, $S_{ST} = S \oplus S^x$.

$S_{SRF} = S \oplus S^x$, where y is a simple expression in S, $y = x$.

The definitions of S_{SNF} and S_{SRF} are identical to the definitions of S_{VNF} and S_{VRF} in Section 1.2 except that "simple expression" is substituted for "variable". Simple expression was defined to closely correspond to the Boolean variable in [14]. The definition of S_{ENF} comes from Section 1.2. So under conditions in that section, $S_{SNF} \rightarrow S_{ENF}$ and $S_{SRF} \rightarrow S_{SNF}$.

ORF and ORF$^+$ are not defined for expressions; therefore, we cannot strictly analyze their relationship to S_{SNF}. However, since ORF and, especially, ORF$^+$, are defined to closely match VRF, we believe that ORO$^+$ detects SNF.

Theorem If the simple expression replaced by 0 or 1 in S_{STF} is the same simple expression negated in S_{SNF}, then $S_{STF} \rightarrow S_{SNF}$.

Proof.

As in [14], the theorem follows if $dS^{a_1}_0 \rightarrow dS^{a_1}_{\bar{a}_1}$ and $dS^{a_1} \rightarrow dS^{a_1}_{\bar{a}_1}$.

The detection conditions for arbitrary stuck-at-0 and simple expression negation faults are $dS^{a_1}_0 = S \oplus S^{a_1}_0$ and $dS^{a_1}_{\bar{a}_1} = S \oplus S^{a_1}_{\bar{a}_1}$.

For simplicity, let $B = (\bar{b}\ \ \bar{b}\ \ \cdots\ \ \bar{b}_m) \wedge \cdots \wedge (\bar{z}\ \ \bar{z}\ \ \cdots \vee \bar{z}_n)$.

$dS^{a_1}_0 = (a\ a\ .\ .a_k\ \ b\ b\ .\ .b_m\ \ \cdots\ \ z\ z\ .\ .z_n) \oplus (0\ \ b\ b\ .\ .b_m\ \ \cdots\ z\ z\ .\ .z_n) = (a\ a\ .\ .a_k \oplus 0) \wedge B = (a\ a\ .\ .a_k) \wedge B$.

$dS^{a_1}_{\bar{a}_1} = (a\ a\ .\ .a_k\ \ b\ b\ .\ .b_m\ \ \cdots\ \ z\ z\ .\ .z_n) \oplus (\bar{a}\ a\ .\ .a_k\ \ b\ b\ .\ .b_m\ \ \cdots\ \ z\ z\ .\ .z_n) = (a\ a\ .\ .a_k \oplus \bar{a}\ a\ .\ .a_k) \wedge B = (a\ .\ .a_k) \wedge B$.

Since $a\ a\ .\ .a_k \rightarrow a\ .\ .a_k$, then $dS^{a_1}_0 \rightarrow dS^{a_1}_{\bar{a}_1}$.

Similarly, for a stuck-at-1 fault, since $dS^{a_1} = S \oplus S^{a_1} = (\bar{a}\ a\ .\ .a_k) \wedge B$, then $dS^{a_1} \rightarrow dS^{a_1}_{\bar{a}_1}$.

Therefore, $S_{STF} \rightarrow S_{SNF}$. Q.E.D.

Now consider MCF and STF. Dropping a simple expression x from a conjunction is the same as setting $x = 1$, which is a stuck-at-1 fault. Dropping x from a disjunction is the same as setting $x = 0$, a stuck-at-0 fault. Therefore, $S_{STF} \rightarrow S_{MCF}$.

Test Generation from Actual Specifications

Actual specifications are generally not in DNF. The mutants of a DNF representation are different from the mutants of the original. One part of our current research is to determine what effect, if any, this difference has on resulting tests.

To illustrate the difference, let S be a specification and S' be the DNF representation of S. Some first order mutants of S cannot be generated from S' since they are higher order mutants of S', and some first order mutants which would be generated from S' are actually higher order mutants of the original S.

For example, if $S = x \rightarrow y$, then $S' = xy\ \ \bar{x}y\ \ \bar{x}\bar{y}$. Setting the first appearance of x to 0 in S' results in $\bar{x}y\ \ \bar{x}\bar{y} = \bar{x}$, which is not a first order mutant of the original formula and is an unlikely error.

Consequently, we apply mutation operators to the unaltered specification, and the theoretical results do not strictly apply.

Suppose OP1 and OP2 are mutation operators and F1 and F2 are their respective fault classes. Suppose also that $S_F \rightarrow S_F$. This suggests that OP1 detects F2, that is, $T_O \subseteq T_O$. However, the implication may be trivially true because S_F is universally false or S_F is universally true. Consider the case where OP1 generates a consistent mutant, e.g. , if the specification is SPEC AG (x | y), then $S = x\ \ y$, and

$dS^x = (x\ \ y) \oplus (1\ \ y) = \bar{x} \wedge \bar{y}$.

$dS^x_{\bar{x}} = (x\ \ y) \oplus (\bar{x}\ \ y) = \bar{y}$.

Therefore, $dS^x \rightarrow dS^x_{\bar{x}}$. However, the mutant generated by setting x to 1 is always true and does not result in a test case.

Similarly, OP1 may not generate a mutant. Suppose an SMV specification contains only one variable, x, of type Boolean, and one clause SPEC AG (x). SRO does not generate any mutants, whereas SNO generates a mutant with $\bar{x}$. This mutant is likely to produce a test case. Even though $S_{SRF} \rightarrow S_{SNF}$, since UT_{SRO} is empty, $T_{SNO} \subseteq T_{SRO}$, and SRO does not detect SNF in this case.

Since the same test case is usually derived from a number of mutant specifications, we hypothesize that the issues mentioned in this section do not significantly affect the results for SMV specifications of considerable size. However,

	SPEC clauses	Boo-leans	Sca-lars	Inte-gers	Total vars
Cruise Control	14	8	3	0	11
Safety Injection	22	1	3	1	5
CPU Stack	21	1	3	0	4

Table 2. Number of CTL Formulas and Variables in Sample Specifications.

	Mutants	U - I Mutants	Unique Traces
Cruise Control	879	116	24
Safety Injection	730	86	21
CPU Stack	924	81	9

Table 3. Number of Mutants and Traces for Specifications.

it is important to experimentally support these theoretical results.

3.2 Empirical Comparison of Mutation Operators

To empirically confirm these results, we developed an extensible tool for generating mutations of SMV specifications, using the SMV parser. It allows us to selectively apply mutation operators. Resulting individual mutations may be left in individual SMV files or combined into a single file for faster model checking. The source code and documentation are available from the authors.

We ran experiments on three SMV specifications to compare the mutation operators in terms of the number of test cases produced and the specification coverage. Table 2 shows the number of CTL formulas and the number of variables in each of the specifications. Here are some additional details:

Cruise Control [3]

Two of the scalar variables have the same domain: {Activate, Deactivate, Resume}. The third has a domain of cardinality 5.

Safety Injection [5]

Two scalar variables have the domain: {On, Off}. The third has a domain of cardinality 3. The integer variable takes values between 0 and 200, but it is only compared with 2 different symbolic constants.

CPU Stack

The scalars have domains with cardinality 3, 4, and 6, respectively.

Empirical Evaluation of Mutation Operators

Table 3 gives the total number of mutants, the number of semantically unique, inconsistent (U-I) mutants, and the number of unique test cases or traces generated by applying the mutation operators to the sample specifications.

Operator	Mutants	CEs	UTs	Coverage
ORO$^+$	405	152	24	100%
ORO	405	152	24	100%
SNO	72	47	21	96.6%
ENO	130	105	21	96.6%
LRO	116	87	14	87.9%
RRO	-	-	-	
MCO	72	40	18	93.1%
STO	144	47	21	96.6%
ASO	12	8	4	62.9%

Table 4. Cruise control example results.

Operator	Mutants	CEs	UTs	Coverage
ORO$^+$	202	99	21	100%
ORO	130	63	17	94.2%
SNO	83	51	15	90.7%
ENO	144	104	15	90.7%
LRO	122	82	10	83.7%
RRO	72	36	10	50.0%
MCO	79	50	13	87.2%
STO	166	51	15	90.7%
ASO	17	17	5	47.7%

Table 5. Safety injection example results.

Operator	Mutants	CEs	UTs	Coverage
ORO$^+$	279	135	9	100%
ORO	279	135	9	100%
SNO	75	52	7	97.5%
ENO	129	100	7	97.5%
LRO	129	46	5	90.1%
RRO	-	-	-	
MCO	109	38	7	97.5%
STO	256	52	7	97.5%
ASO	22	20	4	85.2%

Table 6. CPU Stack example results.

We present details in Tables 4, 5, and 6. As in Table 3, "Mutants" is the total number of mutants generated by each operator, including consistent and duplicate mutants. Since SNO mutants are a subset of ENO mutants, we do not include SNO mutants in the number of mutants in Table 3. Next we give the number of counterexamples, "CEs," found in the SMV runs. "UTs" is the number of unique traces after duplicate traces and prefixes are removed.

We use the specification-based coverage metric introduced in [1]. We exclude all consistent mutants. We also exclude all but one copy of inconsistent mutants which are semantic duplicates of other mutants, e.g., those which always evaluate to the same result. Let N be the number of U-I mutants generated by all operators for a given example. We turn the unique traces from each operator into constrained finite state machines, then SMV finds which mutants are killed. Let k be the number of mutants killed. The coverage is $\frac{k}{N}$.

Results for RRO appear only for Safety Injection, since it is the only example with relational operators.

Discussion

ORO$^+$ generates the largest number of mutants, but provides the same set of test cases as all the operators combined. Consequently, it has 100% coverage.

SNO, ENO, and STO each provide second best coverage. SNO, however, generates significantly fewer mutants.

MCO provides slightly less coverage while generating a small number of mutants. As mentioned in [14], a common implementation error is the failure to validate input data or check preconditions. This is an MCF. Since MCO is designed to detect MCF, its good performance should not be surprising.

LRO generates a large number of mutants and provides good coverage for each example. ASO has low coverage, but generates very few mutants.

In these examples, we found the following relationships between the sets of unique traces:

$$UT_{SNO} \subseteq UT_{ORO}$$

$$UT_{MO} \subseteq UT_{STO}$$

$$UT_{SNO} \subseteq UT_{STO}$$

$$UT_{ENO} \subseteq UT_{SNO}$$

These results agree with the analysis in Section 3.1. In particular, they support the idea that ORO is sufficient to detect faults in ORF, SNF, and ENF. This suggests that SNO and ENO are not needed if ORO is used.

4 Conclusions

It is widely accepted that testing is a crucial, but sometimes overlooked, part of software engineering. Developing adequate test sets is often a labor-intensive and tedious task. A recent method, combining mutation analysis and model checkers, automatically generates complete test sets from formal specifications. In this paper, we report that we refined or invented several useful specification mutation operators for this method and we compared these and other operators.

We found that a combination of Operand Replacement and Relational Operator Replacement mutation operators, ORO$^+$, has the most coverage of all the operators we considered, but generates a large number of mutants. The Simple Expression Negation Operator, SNO, has good coverage, and generates a small number of mutants. The Missing Condition Operator (MCO) has similar coverage to and generates about the same number of mutants as SNO. However, MCO may be preferred since it models missing predicates, a common programming fault. The other mutation operators had poorer coverage or generated more mutants than these three operators.

The theoretical analysis and experimental data are consistent with each other, supporting our claim that these mutation operators are practical for automatically generating complete test sets from specifications.

References

[1] P. E. Ammann and P. E. Black. A specification-based coverage metric to evaluate test sets. In *Proceedings of Fourth IEEE International High-Assurance Systems Engineering Symposium (HASE 99)*, pages 239–248. IEEE Computer Society, November 1999. Also NIST IR 6403.

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

[3] J. M. Atlee and M. A. Buckley. A logic-model semantics for SCR software requirements. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis*, pages 280–292, Jan. 1996.

[4] C. Banphawatthanarak, B. H. Krogh, and K. Butts. Symbolic verification of executable control specifications. In *Proceedings of the* 10 ^{h} *IEEE International Symposium on Computer Aided Control System Design (jointly with the 1999 Conference on Control Applications)*, pages CACSD–581–586, Kohala Coast - Island of Hawai'i, Hawai'i, Aug 1999.

[5] R. Bharadwaj and C. L. Heitmeyer. Model checking complete requirements specifications using abstraction. Memorandum Report NRL/MR/5540-97-7999, U.S. Naval Research Laboratory, Washington, DC 20375, November 1997.

[6] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using model-checking. In *Proceedings 1996 SPIN Workshop*, Rutgers, NJ, August 1996. Also WVU Technical Report #NASA-IVV-96-022.

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

[8] E. M. Clarke, Jr., E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[9] R. A. De Millo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[10] A. Engels, L. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In E. Brinksma, editor, *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems. (TACAS'97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 384–398. Springer-Verlag, April 1997.

[11] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proceedings of the Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Toulouse, France, September 1999.

[12] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, 1(2):156–173, June 1975.

[13] D. R. Kuhn. A technique for analyzing the effects of changes in formal specifications. *The Computer Journal*, 35(6):574–578, 1992.

[14] D. R. Kuhn. Fault classes and error detection in specification based testing. *ACM Transactions on Software Engineering Methodology*, 8(4), October 1999.

[15] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[16] K.-C. Tai. Theory of fault-based predicate testing for computer programs. *IEEE Transactions on Software Engineering*, 22(8):552–562, Aug. 1996.

[17] M. Woodward. Errors in algebraic specifications and an experimental mutation testing tool. *Software Engineering Journal*, pages 211–224, July 1993.