Specification Mutation for Test Generation and Analysis

by
Vadim Okun

Dissertation submitted to the Faculty of the Graduate School
of the University of Maryland Baltimore County
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
2004

ABSTRACT

Mutation analysis is a fault-based testing technique that uses mutation operators to introduce small changes into a program or specification, producing mutants, and then chooses test cases to distinguish the mutants from the original. Mutation operators differ in the coverage they get. They also differ in the number of mutants they generate. Consequently, selecting mutation operators is an important problem whose solution affects the effectiveness and cost of mutation testing.

We use the automated test generation and evaluation method that combines a model checker and mutation analysis. We define a set of mutation operators and implement a mutation generator for specifications written in SMV, a popular model checker.

To select the most effective mutation operators and sets of operators, we compare them using both theoretical and experimental methods. We construct mutation detection conditions and develop a technique to theoretically compare mutation operators. We apply mutation coverage and pairwise coverage metrics to empirically compare the effectiveness of mutation operators.

To detect a fault in a program, a test case must cause the fault to affect the outputs, not just intermediate variables. We develop a method that uses a model checker to guarantee that tests cause visible output failures.

We find that mutation operators form a hierarchy with respect to detection capability; we can skip a test for a mutation from an easier-to-detect mutation operator in the hierarchy, provided that we detect a corresponding mutation from a harder-to-detect operator. Our theoretical technique allows us to prove that the hierarchy applies to arbitrary logic expressions, whereas previous results apply only to logic expressions in disjunctive normal form. Based on analysis and empirical evaluation, we recommend mutation operators and sets of mutation operators that yield good test coverage at a reduced cost.

Our experiments show that specification-based mutation can be applied to test programs; it gets good program-based coverage. Our method for guaranteeing fault visibility is very effective for black-box testing of programs which have a large intermediate state.

This thesis shows that specification-based mutation can be used to economically generate effective tests.

ACKNOWLEDGEMENT

I would like to first thank my dissertation advisor, Yaacov Yesha, for his patience and faith in me, as well as numerous discussions. He allowed me considerable freedom in my research.

I have been very lucky to work with Paul Black. This dissertation would not be possible without him. Paul introduced me to this field and always helped and encouraged me. He taught me a lot about thinking creatively, doing research, and presenting information. Many ideas in this dissertation have originated from discussions with Paul. The ideas in this dissertation were strengthened in discussions with Paul. He thoroughly reviewed my dissertation and made numerous suggestions for its improvement.

Thanks to Paul Ammann for useful discussions and encouragement. He and Paul Black pioneered this area of research.

I thank the members of my excellent dissertation committee (Paul Ammann, Paul Black, Kostas Kalpakis, Sam Lomonaco, Charles Nicholas, and Yaacov Yesha) for providing valuable advice.

I would like to thank Jeff Offutt, a leading expert in mutation testing, for insightful comments about a part of this work and for sharing programs and their faulty versions with me.

Finally, thanks to my parents, Ernst Okun and Nataliya Rosina, for their support.

Contents

Chapter 1

Introduction and Motivation

Few users are satisfied with reliability of their software. Even though the quality assurance budgets of software makers are increasing, program failures with possible data loss remain quite common. Failures are especially dangerous in safety-critical systems, such as aeronautics and medical applications.

A program failure is caused by a *fault*, that is, a defect in the code, informally, a bug. Testing is a way to find faults in software. It is a process of supplying a system under test with some values and making conclusions on the basis of its behavior. A *test case* consists of inputs together with the expected results. Although generating test inputs can be as simple as selecting numbers randomly, deriving the corresponding expected results is often labor intensive.

Testing criteria define, in a quantifiable way, what should be tested and when the objective of testing is achieved. For example, statement coverage requires that every statement in the program is executed at least once during testing. Test sets may be chosen according to a number of different testing criteria.

The testing criteria can be compared based on their relative effectiveness and cost. The *effectiveness* of a criterion is determined by its ability to detect faults. Since the number of faults can be infinite, we may choose to concentrate, on detecting a limited subset of faults; mutation analysis is an example of this approach.

Alternatively, we may concentrate on the behavior instead of the code and attempt to systematically cover the entire domain of a system. *Pairwise coverage* requires that for each pair of inputs, every combination of valid values of these two inputs be covered by at least one test case.

Testing criteria can be classified into program-based and specification-based categories. Program-based (or white-box) testing is based on the code without consideration of design. Thorough white-box testing is prohibitively expensive for large software systems. Additionally, it provides no information about whether the code is doing what it is supposed to be doing [35].

In contrast, specification-based (or black-box) testing derives test cases from the specification of a system. A specification provides valuable information about the intended behavior of the implementation, and therefore about the expected test results [31].

Automated test generation from formal specifications promises to improve our ability to test software that has to be highly reliable, as well as lower the cost of testing off-the-shelf software. A novel way was developed in [3] to automatically produce tests from formal specifications and measure test coverage using a combination of specification-based mutation testing criterion and model checking. The method is described in Section 2.6. Mutation generation is an important part of the method.

Mutation analysis [23] is a method for developing sets of test cases which are sensitive to small syntactic structural changes. A mutation analysis system defines a set of mutation operators. Each operator is a pattern for a small syntactic change; it models a particular class of faults. A mutant is produced by applying a single mutation operator exactly once to the original specification (program). For instance, the insertion mutation operator can replace a Boolean variable with a disjunction of the variable and another Boolean variable. Applying the set of operators systematically generates a set of mutants. If a test set can distinguish a specification from each slight variation, the test set is exercising the specification adequately.

Accordingly, mutation adequacy is a testing criterion which specifies the percentage of the mutants distinguished by the test set. The method in [3] automatically generates a mutation adequate test set, that is, a test set able to detect mutants generated by a chosen set of mutation operators. Mutation operators differ in the number of mutants they generate. They also differ in the coverage they get. Consequently, the cost and effectiveness of mutation testing depend on which mutation operators are used.

This is a study of specification mutation. Although program-based mutation has been extensively studied over the years, previous specification-based mutation work does not systematically compare the effectiveness and cost of different mutation operators and does not give a prescription for which sets of mutation operators should be used.

1.1 Thesis Statement

Specification-based mutation analysis can be used to economically generate effective tests. Comparing mutation operators based on their coverage and the number of mutants generated allows us to select subsets of operators which have lower cost while maintaining high fault detection capabilities of the test set.

To ensure that choice of mutation operators is appropriate in most situations, the comparison must be independent of a particular experimental base. Mutation operators can be compared theoretically, so that the results do not depend on a chosen experimental base; moreover, these results apply to arbitrary logic expressions, not just those in disjunctive normal form.

Mutation operators form a hierarchy with respect to detection capability; we can skip a test for a mutation from an easier-to-detect mutation operator in the hierarchy, provided that we detect a corresponding mutation from a harder-to-detect operator. In practice, tests generated by harder-to-detect operators are either more effective, or likely to be as effective while being less costly, than those generated by easier-to-detect operators.

A practical goal of testing is to reduce the number of faults in the programs corresponding to the specifications. To detect a fault in a program, a test case must cause the fault to affect the program output, not just intermediate variables. A model checker can be used to select tests that cause detectable output failures. Specification-based mutation can be applied to test programs; it gets good program-based coverage.

1.2 Methods Used

To select the most effective mutation operators and sets of operators, we compare them using both theoretical and experimental methods. We give a brief overview of the methods below.

We develop a technique to theoretically compare mutation operators. In this technique, we construct detection condition for a mutation in an expression as a conjunction of origination condition and propagation condition. The origination condition for a mutation is the condition under which the mutation changes the value of the smallest subexpression corresponding to the mutation. The propagation condition is the condition under which the value of the whole expression changes if the value of its subexpression changes. This construction enables us to prove detectability relationships between mutation operators for arbitrary logic expressions, not just those restricted to disjunctive normal form.

To enable empirical comparison of mutation operators and sets of operators, we use the automated test generation and evaluation method developed in [3]; it combines specification mutation analysis and model checking. A model checking specification consists of a state machine description and temporal logic expressions over states and execution paths. The model checker verifies that each temporal logic expression is consistent with the state machine. If there is an inconsistency, the model checker generates a counterexample in the form of a sequence of states, if possible. The method in [3] systematically mutates a model checking specification and uses a model checker to generate counterexamples, then turns them into test cases. We use the method to generate test sets and also to evaluate mutation coverage of test sets. We implement a tool to mutate specifications written in the language of a popular model checker, SMV.

We use pairwise coverage to get an independent indication of the quality of mutation operators and sets of operators.

The method in [3] may generate tests that do not cause faults to affect the outputs of the program corresponding to the specification. We develop a method to generate test cases that cause detectable output failures. In this method, we make a copy of the state machine and mutate the copy. Then the model checker compares the external behavior of the original and mutated state machines. Any counterexamples produced will exhibit differences in the outputs.

We use fault-based (fault seeding) and structural (block, branch, and data-flow) coverage metrics to evaluate the applicability of specification mutation to testing programs corresponding to the specifications.

1.3 Pointers to Published Portions of the Thesis

We define mutation operators in [10] and more comprehensively in Section 3. We describe the mutation generator tool in [9] and in more detail in Appendix A. We experimentally compare mutation operators in [10] and in depth in Section 7. We present mutation detection conditions and use them to analytically compare mutation operators in [61] and in Sections 4, 5, and 6. We describe our method guaranteeing that tests cause detectable output failures in [59] and in more detail in [60] and in Section 8.

1.4 Additional Results

We use our theoretical technique to analyze existing testing methods. In particular, we show that the basic meaningful impact strategy [82] is stronger in that it tests for mutations from a harder-to-detect mutation operator than was claimed by the authors. We also find that if we detect mutants generated by the insertion mutation operator, we get almost perfect pairwise coverage.

1.5 Organization of the Thesis

The thesis is organized as follows. Section 2 presents background information. Section 3 defines mutation operators and some combinations of mutation operators. They are compared using both theoretical and experimental methods.

Sections 4–6 theoretically compare mutation operators. Section 4 defines the mutation conditions and explains our theoretical approach. Section 5 uses the approach to prove the hierarchy of mutation operators. Section 6 applies the mutation conditions and mutation hierarchy to compare several specific mutation categories, to consider some previous empirical observations, and to analyze the basic meaningful impact strategy of Weyuker et al.

Section 7 experimentally compares mutation operators and their combinations in terms of their specification-based coverage. Appendix B presents details of some experiments.

To detect a fault in an implementation, a test case must cause the fault to affect the externally visible outputs. Section 8 presents our approach to guarantee fault visibility. Section 9 evaluates the effectiveness of specification mutation for testing the implementations. Section 10 summarizes contributions of the thesis. Appendix A describes the mutation generator tool.

Chapter 2

Background

We survey research of others on testing criteria, fault visibility, fault classes, mutation analysis, and model checking. Section 2.6 describes the method of test generation and evaluation [3] that combines a model checker and mutation analysis.

2.1 Testing Criteria

A *testing criterion* [31] specifies what properties of a program or specification must be exercised to constitute a thorough test. Examples of testing criteria are statement coverage, branch coverage, multiple condition coverage, and mutation adequacy. Criteria for generating tests from state-based specifications are presented in [58].

2.1.1 Methods of Comparison

Weyuker et. al [83] examined the means of comparing different testing criteria. The most important bases for comparison are effectiveness and cost. Cost is secondary to effectiveness because an ineffective criterion, no matter how cheap, is a poor choice. The effectiveness of a criterion is the extent to which it enables us to detect faults. Since it is usually impossible to enumerate all possible faults, one practical approximation is to measure the ability to detect mutants.

Various methods of comparison have their strengths and weaknesses. First, some methods allow to compare more criteria than others. Second, some methods rank different criteria qualitatively, while others quantitatively. Third, some methods compare with respect to all possible programs and specifications, while others compare the criteria for a particular program.

Subsumption relationship is a widely used method for comparing different testing criteria. A criterion C_1 is said to subsume another criterion C_2 if and only if any test set that satisfies C_1 also satisfies C_2. A subsumption hierarchy for several path selection criteria was developed in [20]. Many criteria based on logical control flow through a program [17] are subsumed by mutation testing [54]. Many criteria are incomparable under subsumption. Even when subsumption orders two criteria, it gives no quantitative measure of their differences in terms of effectiveness [52]. Theoretical comparisons often have these kinds of weaknesses.

Empirical comparisons usually give quantitative measure, however, they depend on the chosen experimental base and it is hard to extrapolate the results.

Some empirical studies use one testing criterion (for instance, mutation adequacy in [82] and [42]) as a standard for comparison of other techniques. Such comparisons tend to favor those criteria which are similar to the criterion used as the standard. This problem can be alleviated by using more than one standard.

```
if (a > 1)
    x = 2;
else
    x = 5;
if (x == b)
    printf("Equal\n");
else
    printf("Not equal\n");
```

Figure 2.1: A Code Fragment Illustrating P-use Coverage.

Since no method of comparison is perfect, we believe that using several different approaches gives the best results. In this thesis, we compare the mutation operators analytically, then compare them empirically using two different metrics: mutation coverage and pairwise coverage.

2.1.2 Overview of Testing Criteria

Zhu et. al [88] list three basic approaches to software testing: structural testing, fault-based testing, and error-based testing. *Structural testing* requires coverage of a particular set of elements in the structure of the program or specification. Two main groups of program-based structural test adequacy criteria are control-flow criteria and data-flow criteria. As the names suggest, they are based on the flow-graph model of program structure.

Examples of control-flow criteria are statement coverage, block coverage, branch coverage, and path coverage [22]. *Statement coverage* reports whether each executable statement is encountered. *Block coverage* counts the branch-free executable code fragments that are exercised at least once. A block may be more than one statement if there is no branching between statements. An expression may contain multiple blocks if there is branching implied in the expression, for instance, an expression with logical connectors. If block coverage is less than 100%, some statements are not exercised by the test set. *Branch coverage*, or *decision coverage*, checks whether logic expressions tested in control structures evaluated to both true and false. *Condition coverage* reports the true or false outcome of each logic subexpression. *Multiple condition coverage* checks whether every possible combination of logic subexpressions occurs. *Modified condition/decision coverage (MC/DC)* requires enough test cases to verify that every condition can affect the result of its encompassing decision. *Path coverage* checks whether every possible path has been followed. Statement coverage is subsumed by decision coverage, which in turn is subsumed by path coverage.

Data-flow testing [68] classifies each occurrence of a variable as either an assignment to the variable or a use of the variable. A good test set must insure that for each possible assignment to a variable, the uses of the variable are exercised. We follow the definitions in [37]; other data flow coverage measures have been proposed, see [20]. *C-use*, or computation variable use coverage, counts the number of combinations of an assignment to a variable and a use of the variable in a computation that is not part of a conditional expression. *P-use*, or predicate variable use coverage, counts the number of combinations of an assignment to a variable and a use of the variable in a conditional expression, and all branches based on the value of the conditional expression.

We use the C code fragment in Figure 2.1, derived from [84], to illustrate P-use coverage. Variables a and b are input variables. Here, for instance, a test case $(a, b) = (2, 2)$ covers the P-use pair consisting of the assignment $x = 2$ and the use of x in $x == b$ where $x == b$ is true, while a test case $(a, b) = (2, 3)$ covers the same P-use pair where $x == b$ is false. Similarly, a test case $(a, b) = (1, 5)$ covers the P-use pair consisting of the assignment $x = 5$ and the use of x in $x == b$ where $x == b$ is true, while a test case $(a, b) = (1, 4)$ covers the same P-use pair where $x == b$ is false.

All-use coverage is the sum of C-use and P-use coverage measures.

Fault-based testing focuses on generating tests to detect faults in software [88, 49, 23, 86]. It can often guarantee the absence of particular faults, which is an important advantage over other testing approaches. We describe mutation analysis [23], a fault-based testing technique, in Section 2.4.

Error-based testing focuses on checking programs at certain error-prone points. An example is boundary testing.

2.1.3 Design of Experiment and Pairwise Testing

When the logic of a program is faulty, a region of the input domain will exhibit failures. Such region faults are more likely to remain undetected if a test set leaves big portions of the input domain uncovered than if test cases are distributed uniformly throughout the input domain. This assumption is supported empirically in [12].

Design of experiment [63] focuses attention on the usage of software and attempts to generate tests that span the entire input domain of a system. Combinatorial testing attempts to cover all k-way combinations of input parameters. In particular, pairwise testing [75] (or 2-way testing) is a specification-based testing criterion which requires that for each pair of input parameters of a system, every combination of valid values of these two parameters be covered by at least one test case. Experiments show that pairwise testing is effective for software [21].

We introduce the notation and terminology used in pairwise testing. Consider a system with $n \geq 2$ input variables. The value u of variable α is denoted by $\alpha.u$. The *pair* of values $\alpha.u$ and $\beta.v$, where α and β are different variables, is denoted by $(\alpha.u, \beta.v)$ or $(\beta.v, \alpha.u)$. A test for the system has n values, one for each input variable. A test is said to *cover* a pair $(\alpha.u, \beta.v)$ if it assigns u to α and v to β. A test set is said to cover a pair if at least one of its tests covers the pair. We use pairwise coverage to evaluate specification-based mutation.

2.2 Fault Visibility

In black-box testing, we neither look at the code nor check the intermediate state of the program, and rely strictly on the results (outputs) for detecting a fault. Accordingly, to detect a fault in a program, we supply the program with some test inputs and compare the results of the execution with the expected results, thus determining whether the results are correct.

A (visible) *failure* is an unacceptable result of execution on some test input; in other words, it is observable incorrect behavior. A *potential failure*, or potential error, is an intermediate incorrect result. In order for a test to detect a fault, the erroneous intermediate values must cause an error in the outputs. In other words, a potential failure must propagate through computations and information flow to produce a visible failure. In black-box testing, if a test case does not cause a visible failure, it does not detect the fault.

There is an extensive body of research on conditions for detecting a fault from the program output [28, 24, 50, 71, 86, 33]. Goradia [34] presents typical situations where a test case causes a potential failure but not a visible failure.

- The faulty state variable does not participate in a computation that affects the output. Consider this code fragment where variable `output` is the only visible outcome:

  ```
  if (condition) {
      output = state_var;
  } else {
      output = 10;
  }
  ```

  If `condition` is `false`, an incorrect value of `state_var` does not impact the output.

- The faulty state variable is used in an operation that affects the output, but the operation may not be sensitive to the error represented by the incorrect state. For example, in an expression `state_var > z`, the value of `state_var` affects its result. However, an incorrect value of `state_var` may yield the correct Boolean value of the relational expression.

- Cancelling errors. The faulty state variable may interact with another faulty state variable or itself and thereby yield a correct state. For example, in an expression `x * y`, if both `x` and `y` have incorrect signs, the result will have correct sign.

- An algorithm may tolerate errors in the values of certain variables. Consider a numeric algorithm which computes the local minimum of a polynomial in a given interval by using an iterative procedure that terminates when a specific convergence criterion is satisfied. At the end of each iteration, it obtains the next approximation by adding the value of a variable `step` to the previous approximation. If the value of `step` is faulty, the algorithm may still converge to the correct result by changing the number of iterations.

In this thesis (Section 8), we present an approach which uses a model checker to guarantee that tests chosen cause visible output failures.

Program mutation testing in its original formulation, often referred to as strong mutation, requires the output of a mutant program to differ from the original. Weak mutation [38], on the other hand, only requires that the execution of a component of the mutant and the original produce different values. In this thesis, when our goal is to guarantee visible failures, we require strong mutation.

The RELAY model [71] defines the conditions under which a fault is detected. First, a potential error originates at the smallest subexpression containing the fault, that is, the subexpression evaluates incorrectly. Then the potential error propagates[1] through computations and information flow. Finally, a failure is revealed in the outputs. The model provides a mechanism for developing failure conditions that guarantee fault detection. In particular, the propagation conditions for Boolean operators are defined. The propagation condition guarantees that a potential failure is not masked out by the computation of a parent operator. We apply the RELAY model to construct the detection conditions for mutations in predicates.

2.2.1 Relevant Work in Protocol Testing

The discipline of protocol conformance testing [11] involves testing an implementation against the protocol specification. Often, the tester has little or no access to the internal states of a protocol implementation because it is running on a remote machine or only its executable code is available (no source code). Tests must be selected to cause a difference in the visible output.

Similarly, one may be interested in testing just a single component of a modular system. The context of the component is the rest of the system. Testing the whole system results in an unnecessarily large test set, while testing the component in isolation raises the problems of test executability (is the test allowed by the context?) and fault visibility (is a fault masked by the context and thus cannot be detected externally?).

These problems were addressed in [62] for the case of a system modeled as a collection of communicating finite state machines, of which one is the specification of the component to be tested; the rest form the context. Testing in context is reduced to testing in isolation by way of computing an approximation of the specification in context. The approximation is a nondeterministic finite state machine model of the component's properties that can be controlled and observed through the context. The behavior of every conforming deterministic implementation is included in the approximation. The tests derived from the approximation are executable and guarantee fault visibility. In our work we rely on a model checker to achieve these goals.

[1] The term "transfer" is used in [71].

Finite state machine models can only specify behaviors within the domain of regular languages. Wang and Liu [81] proposed a test suite generation method for protocols specified by extended finite state machines which associate every transition with an action. The *action* of a transition is any statement or expression, such as an assignment, a conditional expression, input/output, etc. The method assumes that the status of an implementation under test cannot be modified or observed directly, but only by examining the sequences of input and output events. Such observable events are recorded so that the resulting test cases can be applied directly to implementations running on real machines. In this method, an axiom defining the semantics of the actions is associated with each action type. Assertions (preconditions and postconditions) are updated according to the axioms. Assertions consist of a sequence of external (input and output) events appearing along the traversed path, a set of predicates valid at the current state of the extended finite state machine, and variables that need to be observed through the output events in order to confirm correctness of a preselected transition. The method detects any single transition mutant, i.e., a mutant where one transition leads to the wrong state.

2.3 Specification-Fault-Based Testing

An advantage of testing based on specifications is the ”missing path problem” [39]. This occurs when an implementation neglects an aspect of a problem, and there is some section of code that should appear in the program but does not. Since there is no evidence in the code itself for this omission, such errors are very hard to find by analyzing the code alone. However, analysis of specifications can reveal this problem.

Specification-fault-based testing attempts to detect faults in the implementation that are derived from misinterpreting the specification or from faults in the specification [88]. It involves planting faults in the specification.

2.3.1 Test Generation from Logic Specifications

A number of methods have been proposed [28, 74, 76, 82] for generating test cases from specifications represented by logic expressions. The methods usually hypothesize typical classes of faults and derive test sets to detect them. A list of fault classes was summarized in [44]. It includes:

- Variable Reference Fault (VRF) - a Boolean variable x is replaced with another variable y different from x.
- Variable Negation Fault (VNF) - a variable x is replaced with $\bar{x}$.
- Expression Negation Fault (ENF) - a Boolean expression p is replaced with $\bar{p}$.
- Operator Reference Fault - a Boolean operator is replaced with another Boolean operator.
- Incorrect Relational Operators - a relational operator is replaced with a different relational operator.
- Missing Clause Fault - a clause is omitted.

This list is short while the set of possible fault classes is very large. Most testing approaches restrict their attention to single faults, that is, faults that involve only one syntactic change. In particular, mutation operators model the corresponding single fault classes. In Section 3, we define an extensive set of mutation operators.

Boolean operator (BOR) and Boolean and relational operator (BRO) testing strategies were studied in [74] and [76]. In particular, the BOR testing strategy requires a test set to guarantee the detection of Boolean operator faults. A strategy for generating test input from the specifications represented by Boolean formulas was proposed in [82]. The authors claim that the strategy is testing directly for variable negation faults. In this thesis, we show that the strategy is stronger: it tests for stuck-at faults, which are harder-to-detect than variable negation faults.

The analytical results in [82] were obtained for Boolean logic formulas. Specifications for some software can be written in a formal language constrained to Boolean logic expressions. However, the specifications for complex software systems are not exclusively expressed in Boolean logic.

Specifications can be written [65] in:

1. Boolean logic.
2. Arithmetic logic, in which arithmetic equations are included with logic equations.
3. Predicate calculus, which includes quantifying operators such as "for all".
4. Temporal or time-based logic.

Gopal and Budd [32] introduced mutation testing based on specifications in predicate calculus form; we briefly describe their method in Section 2.4.1. We also present temporal logic in Section 2.5.

2.3.2 Kuhn's Fault Hierarchy

Kuhn [43] invented the technique of predicate differences for analyzing the effects of faults in specifications. Briefly, it is as follows. Let S denote a specification predicate hypothesized to be correct and S' a faulty version of it. A test detects the fault if and only if it causes S' to evaluate to a different value than S, formally when $S \oplus S'$. The predicate $S \oplus S'$ is referred to as the detection condition for the fault. The predicate difference is a generalization of the Boolean difference [70] used in hardware testing.

Several researchers [44, 79, 10, 45] used Kuhn's technique to compare fault classes in Boolean specifications restricted to disjunctive normal form, that is, a disjunction of terms. A *term* is a conjunction of literals, a *literal* is an occurrence of a variable or its negation.

Kuhn [44] compared the detection conditions for variable reference fault (VRF), variable negation fault (VNF), and expression negation fault (ENF) and proved, for specifications in disjunctive normal form, that they form a hierarchy with respect to detectability. That is, any test that detects a VRF for some variable also detects a VNF for the same variable, and any test that detects a VNF for some variable also detects an ENF for the expression in which the variable occurs. Tsuchiya and Kikuno [79] proved that a test that detects a missing clause fault (MCF) for some variable will also detect a VNF for the same variable. Tsuchiya and Kikuno also showed that tests that detect MCF may not be able to detect VRF, and vice versa. They also showed that, for terms with more than one variable, any test that detects a VRF for some variable also detects a MCF for the same variable.

Lau and Yu [45] extended the hierarchy to include several other fault classes that can occur in Boolean specifications. They considered literal insertion fault, literal reference fault, literal and term omission faults, literal, term, and expression negation faults. They concluded the following:

- A test case that detects a literal insertion fault where a literal is inserted into a term can also detect a literal reference fault where the same literal replaces a literal in the term.
- A test case that detects a literal insertion fault where a literal is inserted into a term can also detect a term omission fault for the term.
- A test case that detects either a literal reference fault for some literal, a term omission fault for the term containing the literal, or a literal omission fault for the literal can also detect a literal negation fault for the literal.
- A test case that detects a literal negation fault for some literal can also detect a term negation fault for the literal.

- A test case that detects a term negation fault for some term can also detect an expression negation fault for the expression in which the term occurs.

All these results apply to Boolean specifications in disjunctive normal form. We study mutation operators which model fault classes in more general specifications. For instance, the literal insertion fault in [45] is a special case of the clause conjunction operator when specifications are restricted to disjunctive normal form. The clause conjunction operator is defined in Section 3.2.1. Detection condition is an effective and concise analytical tool for studying faults (or mutations) in formal specifications. We refine the detection conditions.

2.4 Mutation Analysis

Underlying program-based mutation testing is the "competent programmer hypothesis" [23], which postulates that programmers write nearly correct programs. Analogously, Ammann and Black [2] propose a "competent specifier hypothesis" stating that analysts write specifications which are likely to be close to what is desired.

Additionally, the "coupling effect hypothesis" [23] states that test data which kill simple first order mutants is also likely to kill higher order mutants, that is, those produced by making simultaneous changes. Thus the coupling effect hypothesis justifies neglecting multiple faults during mutation testing. Offutt's empirical study [55] with several programs found strong support for this hypothesis: any test set that detects all single faults also detects almost all double faults.

Morell [51] defined "fault coupling" as a situation in which a test set can detect faults when they occur in isolation, but not when they occur in combination. This meaning of the term "coupling" is the reverse of that used in the mutation testing literature. Thus, a low incidence of fault coupling implies validity of the coupling effect hypothesis, and vice versa. Morell's definition supports the focus on faults as local objects that may interact with one another. Wah's theoretical study [80] modeled programs as finite functions, and concluded that fault coupling occurs infrequently.

Researchers studied mutation operators for several programming languages. For example, Offutt et. al [53] defined an extensive set of mutation operators for the Ada programs. The mutation operators are separated primarily on the basis of what types of lexical elements are modified.

Mutation analysis is expensive because of a large number of mutants generated. Selective mutation is mutation without the operators that create the most mutants [47, 56]. We propose combinations of mutation operators in Section 7.4.

2.4.1 Early Uses of Specification Mutation

Gopal and Budd extended program-mutation testing to specification-mutation testing in [32]. They considered specifications in predicate calculus form. Figure 2.2 presents a typical specification. There, P and S represent conditions on the input variables prior to execution ("input conditions"), Q, R, T, and U represent conditions on the output after program execution ("output conditions").

The following process is used to evaluate a specification. Given a test input, the program is executed on the input to obtain its output. The input and output values together form a test case. Using the test case generated on the previous step, the specification is "executed" by evaluating each input condition in turn. When an input condition is satisfied by the test input, each of the associated output conditions is evaluated. If any output condition is falsified by the test case, then an error in the program, or specification, or both is discovered and has to be corrected. If all associated output conditions are satisfied, then the test case and the specification match. The test inputs are selected by a method similar to the basic meaningful impact strategy [82] which is described in Section 6.4.

```
if P then
    Q
    R
else if S then
    T
else
    U
```

Figure 2.2: A Sample Predicate Calculus Specification.

After the specification is evaluated, a set of mutation operators is applied to the specification to generate mutant specifications. Each mutant is considered in turn. A test case is successful in eliminating a mutant if there exists an input condition that is satisfied, but for which one or more output conditions is falsified. The tester continues to supply test cases until all the mutants are removed, or until those which remain are equivalent to the original specification. Additionally, while supplying tests to eliminate the mutants, the tester may discover an error in the program.

This method relies on having a working implementation, as the program must be executed in order to generate test output.

Woodward [85] investigated mutation operators for algebraic specifications. The set of mutation operators was defined based on an analysis of errors in specifications made by students.

Woodward considered algebraic specifications as term-rewriting systems. A specification is compiled into the executable code. When the executions of the original specification and a mutant on a given test case generate two different outputs, the mutant is detected and regarded as dead. Test adequacy is measured without executing the program.

We describe the specification-based mutation coverage metric introduced in [2] in Section 2.6.2.

2.5 Temporal Logic and Model Checking

Temporal logic [66] has been used as a formal tool in both artificial intelligence and software engineering [29]. We are interested in temporal logic because it is commonly used as a specification language for model checkers.

Temporal logic is an extension of classical logic for systems that evolve with time. It is used to reason about propositions which may not be true or false once and for all, but which change their truth values through time. The properties such as “Eventually it will be the case that p”, “It will always be the case that p”, “As soon as q is true, it will be the case that p”, etc. can be compactly specified in temporal logic.

Temporal logic is suitable for reasoning about concurrent systems [64]. There are several kinds of temporal logics, two most commonly used are linear and branching [25]. In linear logic, at each moment there is only one possible future. In branching logic, at each moment time may split into alternate courses representing different possible futures.

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. A common logic for model checking is the branching-time Computation Tree Logic (CTL) [19], which extends propositional logic with certain temporal operators. Conceptually, a model checker visits all reachable states and verifies that the temporal logic expressions are satisfied over all paths. (In practice, powerful symbolic computations find equivalent results.) When an expression is not satisfied, the model checker attempts to generate a counterexample in the form of a trace or sequence of states.

Typical formulas in CTL include the following:

- `AG safe`: All reachable states are safe.
- `AG (request → AF response)`: A request is always followed by a response sometime in the future.

We use SMV, a CTL symbolic model checker [48]. In SMV, a specification consists of one or more modules. One module, named `main`, is the top level module in SMV, serving a role similar to that of the function `main` in C programs. Figure 2.3 presents a short SMV example.

```
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)
```

Figure 2.3: An SMV Example.

The model is a Kripke structure, whose state is defined by a collection of state variables. The transition relation of the Kripke structure, and its initial state, are determined by a collection of parallel assignments introduced by the keyword ASSIGN.

A Kripke structure consists of a set of states, a set of transitions between states, and a function that labels each state with a set of properties that are true in this state. Paths in a Kripke structure model computations of the system [18].

In Figure 2.3, `request` is a Boolean input variable, `state` is a scalar variable 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 specification of the system appears as a formula in CTL under the keyword SPEC. The CTL formula in Figure 2.3 states that whenever there is a `request`, state will eventually become `busy`.

SMV also provides the keyword DEFINE, analogous to macro definitions.

By default, all of the assignment statements are executed in parallel and simultaneously. However, it is possible to have a collection of processes whose actions are interleaved in the execution sequence. This is done by preceding an instance of a module with the keyword `process`.

As explained earlier, the transition relation can be specified implicitly as a collection of parallel assignments. It can also be specified explicitly, using the keyword `TRANS`, as a formula in terms of the current and next values of state variables. Similarly, the set of possible initial states can be specified, using the keyword `INIT`, as a formula in terms of the current state variables.

2.6 Software Testing and Model Checking

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 [15]. In addition to verifying properties of software, model checking is being applied to test generation and test coverage evaluation [3, 14, 26, 30, 36, 69].

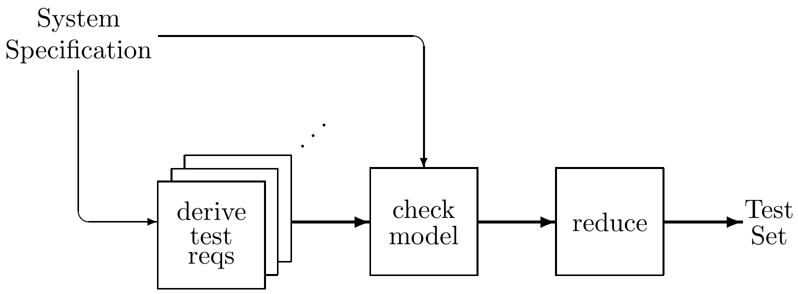


Figure 2.4: Automated Test Generation Method.

In both uses, one begins with selection of a test criterion. Some specification-based test criteria are conjunctive complementary closure partitions [14], branch coverage [30], and mutation adequacy [1]. The overview of the test generation method in [1] is shown in Figure 2.4; we briefly describe the method below.

One applies the chosen testing criterion to the specification to derive test requirements, i.e., a set of individual properties to be tested. To use a model checker, these requirements must be represented as temporal logic formulas; "reflection" [1, 2] may be used for this purpose, see Section 2.6.1.

To generate tests, the requirements must be negative requirements, that is, they are considered satisfied if the corresponding formulas are inconsistent with the state machine. They must also be of a form that a single counterexample demonstrates the inconsistency (exhaustive enumeration is needed to show inconsistency of an existential requirement). For instance, if the criterion is a simple state coverage, the negative requirements are that the machine is never in state 1, never in state 2, etc.

When the model checker finds an inconsistent formula, it produces a counterexample. Again, for state coverage, a counterexample gives stimulus to put the machine in state 1 (if it is reachable), another to put the machine in state 2, etc. Since one counterexample can satisfy more than one requirement, the set of counterexamples may be reduced. Since counterexamples have both stimulus and expected values, they may be automatically turned into executable tests.

In addition to test generation, model checkers can also be used to evaluate coverage of a test set. In this approach, each test is turned into an execution sequence, and the model checker determines which requirements are satisfied by the execution.

2.6.1 Reflection

Reflection represents the state machine in temporal logic. Suppose the state machine description contains a case statement:

```
next(x) := case
    b1 : v1;
    b2 : v2;
    ...
    bN : vN;
esac;
```

In this statement, the *guard* b1 is first evaluated; if it is true, the *target* v1 is the next value for x. To allow for nondeterminism, v1 may be a set of values. If b1 is false, b2 is evaluated, and so forth. "b1 : v1" is called a *guarded command.*

Two forms of reflection were proposed in the literature; we call them *direct reflection* and *guard reflection.* The original description of the specification-based test generation method uses direct reflection [2], which expresses the guarded command “b1 : v1” in temporal logic as follows:

```
SPEC AG (b1 -> AX (x = v1))
```

If v1 is a set, we write “x in v1” instead of “x = v1”. To express the second case, we write:

```
SPEC AG (!b1 & b2 -> AX (x = v2))
```

and so forth. When the case statement is used for specifying the current value of a variable, as in

```
y := case ... esac;
```

the next step operator, AX, is omitted. While easy to apply, direct reflection has some practical limitations; we discuss one of them in Section 7.3. Since the temporal logic formulas derived using direct reflection are positive requirements, in the next step we apply mutation to turn them into negative requirements.

Guard reflection was proposed in [1]. We do not describe the details and derivation of this approach here. In short, guard reflection expresses the guarded command “b1 : v1” in temporal logic as follows:

```
SPEC AG (b1 <-> b1)
```

then applies mutation to the second occurrence of b1. This requires satisfaction of two conditions:

- The guards are a partition, that is, they are pairwise disjoint and their union is universally true. *Expoundment*, a process that makes implicit parts of a specification explicit, helps achieve this. Expoundment is explained in [2].
- The targets are pairwise disjoint, that is, if two guards have the same value for a target, the guards are joined into one guard.

If mutation changes b1 to b1’, it was shown in [1] that under the above conditions, the CTL formula

```
SPEC AG (b1 <-> b1’)
```

is a satisfactory implementation for mutations to b1 in the state machine, assuming that each trace includes one additional state beyond the counterexample to the CTL formula. We use guard reflection whenever possible in our experiments.

2.6.2 Specification-Based Mutation

In the specification based mutation analysis scheme in [3] mutation operators are applied to the state machine or the temporal logic expressions yielding a set of mutant specifications. If the temporal logic expressions are mutated and the state machine is unchanged, the test cases are instances that a conforming implementation must pass. Such tests are called *passing tests.* If instead the state machine is mutated, an implementation conforming to the original specification must fail the test cases. Accordingly, such tests are called *failing tests.* In this thesis we focus on the passing tests.

In the next step, the model checker processes the mutated temporal logic expressions. Mutants represent negative requirements, so they can be used for both test generation and evaluation. When the model checker finds an inconsistency, it generates a counterexample.

The set of counterexamples is reduced by eliminating duplicates and also counterexamples which are ”prefixes” of other, longer counterexamples. The counterexamples contain both stimulus and expected values so they may be automatically converted to complete test cases. For a given set of mutation operators, the procedure in [3] generates a mutation-adequate set of test cases.

Mutations of logic expressions can be consistent or inconsistent with the state machine [2]. A consistent mutant is a temporal logic formula that is true over all possible traces defined by the state machine. Consistent mutants are not useful for model-checking mutation analysis. Inconsistent mutants are either falsifiable (demonstrably inconsistent) or nonfalsifiable (cannot be shown inconsistent with any single trace from the state machine).

2.7 Higher-Level Specifications

SMV's description language is at too low a level for wide-spread use. A popular system must extract state machines from higher level descriptions such as SCR specifications [4], MATLAB stateflows [6], or UML state diagrams. Conversion of UML specifications to SMV is discussed in [77].

State-based specifications, such as SCR, describe the software in terms of states and transitions, they define *preconditions* and *triggering events*. Preconditions must be satisfied in order for a transition to be taken. Triggering events are changes in variable values that cause the transition to be taken. The values of the triggering events before transitions are called *before-values* and the values after transitions are called *after-values*. The state that immediately precedes the transition is the *pre-state* and the state that immediately follows the transition is the *post-state*. In Section 7, we describe the automobile cruise control example using SCR notation.

Chapter 3

Specification Mutation Operators

Many different types of faults can occur in software. (We listed some fault classes in Section 2.3.) Accordingly, software can be mutated in a variety of ways. The choice of mutation operators affects the cost (number of mutants and number of tests) and effectiveness (ability to detect faults) of mutation testing. We compare mutation operators in the following sections. But first, in this section, we define mutation operators. We use the following overall guiding principles, influenced by [85], to formulate mutation operators:

1. Mutation categories should model potential faults.

   It is important to recognize different types of faults. In fact, each mutation operator is designed to model faults belonging to the corresponding fault class.

2. Only simple, first order mutants should be generated.

   These mutants are produced by making exactly one syntactic change to the original specification. This restriction is justified by the "coupling effect" hypothesis which says that the test sets that detect simple mutants will also detect more complex mutants.

3. Only syntactically and semantically legal mutants should be generated.

   Some mutations may result in an illegal expression, such as division by 0. Such mutants should not be generated.

4. Do not produce too many mutants.

   This includes some practical restrictions. For example, we do not replace a relational connector with its opposite, since that is the same as negating the expression. We note these restrictions while defining the mutation operators.

3.1 Definitions

We use the following common mathematical notation throughout the thesis:

- A horizontal line above an operand represents negation (for example, $\bar{a}$).
- $\vee$ denotes disjunction, $\wedge$ represents conjunction, $\rightarrow$ stands for implication, $\oplus$ represents exclusive-or, $\leftrightarrow$ stands for equivalence.
- 1 and 0 are used to denote "true" and "false," respectively.

Among the Boolean operators, negation has the highest precedence, and $\wedge$ has higher precedence than other binary operators. Occasionally, when clear from the context, $\wedge$ is omitted.

When presenting SMV specifications, we use SMV syntax: ! stands for negation, | and & represent disjunction and conjunction, respectively, and != corresponds to $\oplus$.

CTL has two kinds of atoms: variables and symbolic constants. Variables may be Boolean, scalar, integer, user defined modules or an array of any of the above. The value of a scalar variable is drawn from a finite set of constants. An integer variable takes values from an integer range. An SMV specification may also contain symbolic constants defined by the user to represent integers. CTL formulas can have other kinds of constructs, such as case statements, however, those are uncommon and are not generated by reflection, so we do not consider them. Reflection is explained in Section 2.6.1.

We apply mutation to predicates contained within CTL formulas, so we define predicates formally.

A *clause*[1] is one of the following, possibly negated.

- A *Boolean variable*.
- A *scalar expression token*1 *op token*2, where *op* is either = or ≠, *token*1 and *token*2 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 *relational expression* of the form E *op* F, where E and F are arithmetic expressions and *op* is one of $<, \leq, =, \neq, >$, or $\geq$.

A *compound predicate* consists of one or more binary Boolean operators and their operands, and possibly negation operators and parenthesis.

A *predicate*[2] is either a clause or a compound predicate.

For example, $x < 5$ is a clause, and $((x < 5) \vee (y > x)) \wedge (\overline{f \vee g})$ is a compound predicate.

If a clause appears more than once in a predicate, we consider each occurrence to be a distinct clause. This affects our definition of mutation operators as explained in Section 3.2.

A *Boolean formula* consists of Boolean variables and possibly Boolean operators and parenthesis. In other words, a Boolean formula is a predicate with no relational or scalar expressions.

3.2 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 clause fault class can be generated by a missing clause operator (MCO). Note that the abbreviation of the mutation operator ends in O. Below we define mutation operators corresponding to common fault classes.

Abstractly, mutation operators are independent of any particular specification notation. Here we present examples for predicates. Illustrative mutants for each operator are shown in Table 3.1 using SMV notation.

We choose not to assign an abbreviation to every mutation operator since they are hard to remember anyway. We only name those which are necessary for further presentation.

For some mutation operators, we also list their *suboperators*: the statement "M_1 is a suboperator of M_2" means that the set of mutants produced by M_1 is a subset of the set of mutants produced by M_2.

We do not consider mutations of temporal operators because these mutations represent very big semantic change, so the resulting tests tend to be trivial to detect.

We first list mutation operators that involve a clause, then later list mutation operators that affect compound predicates. In the following mutation operators, a clause is replaced with a (possibly empty) predicate.

[1] The terms *condition* or *simple predicate* are also used in the literature [74].

[2] Sometimes we use the term *expression*.

3.2.1 Mutations of Clauses

On the most abstract level, there are four categories of mutation operators affecting a clause.

- Clause Reference Operator (CRO).

  Replace a clause c with another clause, d. For example, replace the specification $(x < 5) \vee (y > 3)$ with $(z > 4) \vee (y > 3)$.

- Clause Negation Operator (CNO).

  Replace a clause c with its negation $\bar{c}$.

- Clause Insertion Operator (CIO). Insert a clause d, that is, replace a clause c with $c \circ d$, where d is another clause, $\circ$ is either conjunction or disjunction. There are two suboperators of this operator.

  – Clause Conjunction Operator (CCO).
    Replace a clause c with $c \wedge d$.

  – Clause Disjunction Operator (CDO).
    Replace a clause c with $c \vee d$.

- Missing Clause Operator (MCO).

  Omit a clause. For instance, replace the specification $c \wedge d \vee e$ with $c \vee e$.

A clause is more general than a Boolean variable, but Boolean formulas are often used for formal specification of real-world systems [46]. For this reason, testing based on Boolean specifications is often studied exclusively [82, 16]. The results in this thesis can easily be adapted to Boolean specifications since many mutation operators in Boolean formulas are special cases of the mutation operators listed above. In particular, in case of Boolean specifications or when only Boolean variables are involved in the substitution, CRO, CNO, and CIO become variable reference operator (VRO), variable negation operator (VNO), and variable insertion operator (VIO), respectively.

CRO may produce a very large number of mutants. Additionally, it generates some higher order mutants such as replacing a Boolean variable with a relational expression. Therefore, except when restricted to Boolean specifications, it should not be used for test generation. For this reason, we consider the practical suboperators of CRO separately. We do not evaluate clause insertion operator empirically, except for Boolean formulas in Section 7.6, so we do not consider its suboperators separately.

Suboperators of Clause Reference Operator

- Operand Reference Operator (ORO).

  Replace an operand, that is, a variable, a constant, or an array subscript, with another syntactically legal operand. We give details later when we present suboperators of ORO.

- Relational Connector Reference Operator (RRO).

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

- Arithmetic Connector Reference Operator.

  Replace an arithmetic connector $(+, -, *, /, mod)$ with another arithmetic connector when appropriate. In many cases such replacement will result in an illegal expression, the most obvious being division by zero. The mutation operator is assumed to recognize such exceptional cases. For our experimental base, the arithmetic connector replacement operator generated very few mutants.

- Off-By-1 Operator (OFO).

  In a relational expression $E_1 operator E_2$, replace the arithmetic expression E_2 with $E_2 + 1$ and with $E_2 - 1$.

- Stuck-At Operator (STO).

  Stuck-at-0 mutation operator replaces a clause with 0; stuck-at-1 replaces it with 1.

Suboperators of Operand Reference Operator

ORO consists of the suboperators presented below:

- Replace a variable with another variable of a compatible type.

  Replace a Boolean variable with every other Boolean variable. Replace a scalar variable x with every other scalar variable having the same domain as x. Replace an integer variable with another integer variable.

- Replace a variable with a constant.

  Replace a scalar variable with every constant from its domain. Replace an integer variable x with every symbolic constant defined by the user to represent a number in the range of x.

- Replace a constant with a variable.

  Replace a constant c from the domain of a scalar variable with every scalar variable x, such that x has c in its domain. Replace a symbolic constant c representing a number in the range of an integer variable with every integer variable x, such that c belongs to the range of x.

- Replace a constant with another constant.

  Replace a constant c in domain D of a scalar variable with every other constant in D. Replace a symbolic constant representing a number in the range R of an integer variable with every other symbolic constant representing a number in R.

  This does not replace an operand if it results in a constant ($c1$ *operator* $c2$) or reflexive (x *operator* x) expression, since an equivalent mutant is produced by the Stuck-At mutation operator. Also, this does not replace a number from integer variable's range with another number, since this may result in too many mutants, and off-by-1 mutation operator represents many typical mutants of this class.

- Array Index Reference Operator.

  Replace an array index with upper and lower array bounds. Replace the index with previous and next integer numbers.

  This does not replace an array index if the previous or next integer number is outside the array bounds.

One Mutation Replaces One Clause Occurrence

Most of the mutation operators involve replacing a clause. Even though the same clause may occur more than once in an expression (for example, y occurs twice in $xy \vee \overline{(y \vee z)}$), a single mutation is a change to just one of the occurrences, not to all of them simultaneously. These mutation operators correspond closely to faults that may occur in software specifications, where one occurrence of a clause or a variable may be replaced as a result of an error while another occurrence is correct. This is in contrast with hardware design, where, for example, a stuck-at-0 fault on a line of a logic circuit results in all occurrences of the corresponding clause being replaced with 0. So whenever we refer to a clause or variable in an expression, we mean a single occurrence of a clause or variable. This approach is also taken by [82, 79, 45].

3.2.2 Mutations of Compound Predicates

The above mutation operators replace a clause. We also introduce the operators that replace a (possibly compound) predicate:

- Expression Negation Operator (ENO).

  Replace an expression X with $\overline{X}$.

  This does not negate temporal expressions, such as AG and EF, since SMV does not produce useful counterexamples from such mutants.

- Missing Expression Operator (MEO).

  Omit a predicate. MEO includes both where a clause is missing and where a compound predicate is missing.

- Logical Connector Reference Operator.

  Replace a Boolean connector with another connector, e.g., replace $x \wedge y$ with $x \vee y$.

  We define the following versions of the logical connector reference operator.

  - LRO$_1$

    Replace one of the three Boolean connectors ($\vee$, $\wedge$, $\rightarrow$) with the other two Boolean connectors.
  - LRO$_2$

    Replace $\vee$ with $\oplus$, $\wedge$ with $\leftrightarrow$, $\rightarrow$ with $\leftrightarrow$. The choice of the new connectors is explained in Sections 6.3 and 7.4.
  - LRO

    Both LRO$_1$ and LRO$_2$, that is, perform all the mutations of LRO$_1$ and LRO$_2$.

- Associative Shift Operator (ASO).

  Change the associativity of terms. For example, replace $(ab) \vee c$ with $a(b \vee c)$.

Table 3.1 contains mutants generated from a predicate "x & (y < 100) → (z = On)", where x is a Boolean variable, y is an integer, z is a scalar. We present the mutants using SMV notation. We list the mutation operators in Table 3.2.

3.3 Combinations of Mutation Operators

It is unlikely that using any single mutation operator can be as effective as using all operators. On the other hand, using all mutation operators results in a very large number of mutants. "Law of diminishing returns" [73] suggests that a combination of a small number of mutation

Operator	Example Mutants
ORO	$x \,\&\, (y < 100) \rightarrow (z = \underline{\text{Off}})$
CNO	$\underline{!}x \,\&\, (y < 100) \rightarrow (z = \text{On})$
ENO	$\underline{!(}x \,\&\, (y < 100)\underline{)} \rightarrow (z = \text{On})$
LRO_1	$x \,\&\, (y < 100) \;\underline{\mid}\; (z = \text{On})$
LRO_2	$x \,\&\, (y < 100) \;\underline{\leftrightarrow}\; (z = \text{On})$
MCO	$\underline{\hphantom{xx}}\; (y < 100) \rightarrow (z = \text{On})$
MEO	$\underline{\hphantom{xx}}\; (z = \text{On})$
STO	$x \,\&\, \underline{1} \rightarrow (z = \text{On})$
ASO	$x \,\&\, \underline{(}(y < 100) \rightarrow (z = \text{On})\underline{)}$
RRO	$x \,\&\, (y \;\underline{<=}\; 100) \rightarrow (z = \text{On})$
OFO	$x \,\&\, (y < \underline{99}) \rightarrow (z = \text{On})$

Table 3.1: Mutation Operators and their Illustrative Mutants.

CIO	Clause Insertion Operator
CCO	Clause Conjunction Operator
CDO	Clause Disjunction Operator
CRO	Clause Reference Operator
ORO	Operand Reference Operator
STO	Stuck-At Operator
MCO	Missing Clause Operator
MEO	Missing Expression Operator
CNO	Clause Negation Operator
ENO	Expression Negation Operator
LRO	Logical Connector Reference Operator
ASO	Associative Shift Operator
RRO	Relational Connector Reference Operator
OFO	Off-By-1 Operator

Table 3.2: Mutation Operators.

operators can be nearly as effective as all mutation operators combined while producing far fewer mutants.

Therefore, we introduce some combinations of mutation operators. Applying a *combination (or a set) of mutation operators* to a specification means applying each individual mutation operator to the original specification, then collecting all the mutants generated by all individual operators.

We consider each mutation operator combined with LRO_2. We justify this choice in Sections 6.3 and 7.4. A combination of ORO and LRO_2 operators is called ORL, a combination of STO and LRO_2 operators is called STL, and so forth. Note that the abbreviation of an individual mutation operator ends in O, and the abbreviation of the combination of the operator with LRO_2 ends in L.

ORO, RRO, and OFO are all suboperators of CRO. However, their application domains are different. While ORO applies to Boolean variables and scalar expressions, RRO and OFO apply to relational expressions. Thus applying ORO exclusively may miss some faults in the relational expressions, while applying RRO and OFO may miss faults in Boolean variables and scalar expressions. Therefore, we introduce a combination of ORO, LRO_2, RRO, and OFO; we call it ORL^+.

Chapter 4

Mutation Conditions

In this section, we first define mutation origination and propagation conditions, then use them to compute mutation detection condition. In the following sections, we will use these mutation conditions to theoretically compare mutation operators.

Since we use the following identities throughout the rest of the thesis, we present them here together. For any predicates f, g, h:

$$\begin{aligned}
f \wedge h \oplus g \wedge h &= (f \oplus g) \wedge h && (4.1)\\
f \wedge h \oplus h &= \bar{f} \wedge h && (4.2)\\
(f \vee h) \oplus (g \vee h) &= (f \oplus g) \wedge \bar{h} && (4.3)\\
(f \vee h) \oplus h &= f \wedge \bar{h} && (4.4)\\
f \oplus h \oplus \bar{f} \oplus h &= 1 && (4.5)\\
(f \vee g) \oplus (f \wedge g) &= f \oplus g && (4.6)\\
((f \wedge g) \rightarrow g) &= 1 && (4.7)
\end{aligned}$$

Identities (4.2) and (4.4) follow from (4.1) and (4.3), respectively. With these identities at hand, we can analyze mutation conditions for various mutation operators.

4.1 Origination Condition

Suppose X is the smallest subpredicate of a specification S corresponding to a mutation, that is, X is replaced with a predicate E. Then the *origination condition* for the mutation is $X \oplus E$, in other words, E evaluates to a different value than X.

For example, an off-by-1 mutation may replace a specification $S = (x < 5) \vee a$ with $(x < 4) \vee a$. Then, the smallest subpredicate of S corresponding to the mutation is the clause $(x < 5)$, and the origination condition is $(x < 5) \oplus (x < 4)$ or $x = 4$. On the other hand, suppose that a logical connector reference mutation replaces the disjunction with a conjunction giving $(x < 5) \wedge a$. Then, the smallest subpredicate of S corresponding to the mutation is S itself, and the origination condition is $((x < 5) \vee a) \oplus ((x < 5) \wedge a)$ which is $(x < 5) \oplus a$.

4.2 Propagation Condition

In what cases will the value of a predicate be affected if one part is mutated? Concretely, if R is some predicate such as $P \wedge Q$, $P \vee Q$, or $P \oplus Q$, what value of Q will let a change in the value of P lead to a change in the value of R? For completeness, we include the case of $R = \overline{P}$. Formally, let $R = op(P, [Q])$ denote either $R = \overline{P}$ or $R = P \circ Q$, where $\circ$ is a binary Boolean connector.

i	P_i	Q_i	op_i	$\widehat{Q_i}$
1	$\overline{vzw}$	$(x \leftrightarrow y)w$	$Q_1 \vee P_1$	$\overline{Q_1}$
2	vzw	none	$\overline{P_2}$	1
3	z	vw	$P_3 \wedge Q_3$	Q_3

Table 4.1: Computing the Propagation Condition for Clause z in $(x \leftrightarrow y)w \vee (\overline{vzw})$.

The *propagation condition* guarantees that the value of R will change if the value of P changes. Denote the propagation condition for *op* as

$$\widehat{Q} = op(1, [Q]) \oplus op(0, [Q])$$

An alternate but equivalent definition of $\widehat{Q}$ is

$$\widehat{Q} = op(P, [Q]) \oplus op(\overline{P}, [Q])$$

For instance, when $R = P \vee Q$, the propagation condition is

$$(P \vee Q) \oplus (\overline{P} \vee Q) = \overline{Q} \quad \text{by (4.3)}$$

Using identities (4.1), (4.3) and (4.5), the propagation conditions for fundamental Boolean operators are as follows:

$$\widehat{Q} = \begin{cases} 1 & \text{if } R = \overline{P} \text{ or } R = P \oplus Q \\ Q & \text{if } R = P \wedge Q \\ \overline{Q} & \text{if } R = P \vee Q \end{cases}$$

Propagation conditions for the other binary Boolean operators fall into one of the three categories above, since they can be expressed using the fundamental operators without duplicating the clause occurrences involved. That is, $P \rightarrow Q = \overline{P} \vee Q$ and $P \leftrightarrow Q = \overline{P} \oplus Q$.

More generally, we can define the propagation condition for a subpredicate X of some larger predicate. It guarantees that a mutation in X is not masked by the computation of parent expressions. In other words, it is the condition under which the value of specification S will change if the value of its subpredicate X changes.

Let $P_0, \ldots, P_n$ be predicates, such that $S = P_0$, $P_{i-1} = op_i(P_i, [Q_i])$, $i = 1 \ldots n$, $X = P_n$. The series of predicates P_i can be seen as the path in the expression tree of S from the root to X. Each Q_i is the subtree on the branch which is not on the path. The propagation condition for a mutation in X is the conjunction of the propagation conditions for each op_i:

$$\frac{dS}{dX} = \widehat{Q_1} \wedge \widehat{Q_2} \wedge \cdots \wedge \widehat{Q_n}$$

Suppose a specification

$$F = (x \leftrightarrow y)w \vee (\overline{vzw}) \tag{4.8}$$

has a clause reference mutation where z is replaced with x. The propagation condition for a mutation in z can be computed from Table 4.1. It follows that

$$\begin{aligned} \frac{dF}{dz} &= \widehat{Q_1}\widehat{Q_2}\widehat{Q_3} = \overline{Q_1} \wedge Q_3 \\ &= \overline{(x \leftrightarrow y)w} \wedge vw = (\overline{x \leftrightarrow y} \vee \overline{w})vw = (x \oplus y)vw \end{aligned}$$

In other words, v and w are true, and y evaluates to the same value as $\bar{x}$. Putting these values back in F and simplifying, we get $(x \leftrightarrow \bar{x})1 \vee (\overline{1z1}) = \bar{z}$, which is sensitive to any change in the value of z.

There may be more than one occurrence of the same clause in a predicate, for instance, variable w occurs twice in (4.8). This makes the notation $\dfrac{dF}{dw}$ ambiguous. However, the concept of clause replacement is unambiguous since each occurrence is considered to be a distinct clause and a mutation is a change to just one of the occurrences. Rather than use an awkward but unambiguous notation, we trust that the reader will understand that the claims made in this thesis have to do with replacing one clause at a time, never several clauses simultaneously.

It turns out that, given two predicates R and P on a path in the expression tree of specification S, such that R is an ancestor of P on the path, if the propagation condition for P is satisfied, then the propagation condition for R is guaranteed to be satisfied. This is stated formally as Lemma 1.

Lemma 1 *Let R be a subpredicate of predicate S. If P is a subpredicate of R, then*

$$\frac{dS}{dP} \rightarrow \frac{dS}{dR}.$$

Proof. Let $P_0, \ldots, P_k, \ldots, P_n, 0 < k < n$, be predicates, such that $S = P_0$, $P_{i-1} = op_i(P_i, [Q_i])$, $i = 1 \ldots n$, $R = P_k$, $P = P_n$. Then

$$\begin{aligned}
\frac{dS}{dP} &= \widehat{Q_1} \wedge \cdots \wedge \widehat{Q_k} \wedge \widehat{Q_{k+1}} \wedge \cdots \wedge \widehat{Q_n} \\
\frac{dS}{dR} &= \widehat{Q_1} \wedge \cdots \wedge \widehat{Q_k}
\end{aligned}$$

In view of (4.7), the Lemma holds. Q.E.D.

We use Lemma 1 to prove Theorem 2 in Section 5.

4.3 Detection Condition

The notation $S^{\mathrm{X}}_{\mathrm{E}}$ signifies that a subpredicate X of specification S is replaced with a predicate E. Kuhn's original definition [44] of the detection condition for the mutation is $dS^{\mathrm{X}}_{\mathrm{E}} = S \oplus S^{\mathrm{X}}_{\mathrm{E}}$, in other words, $S^{\mathrm{X}}_{\mathrm{E}}$ evaluates to a different value than S. For example, if $E = \bar{X}$, an expression negation mutation, the detection condition is $dS^{\mathrm{X}}_{\bar{\mathrm{X}}} = S \oplus S^{\mathrm{X}}_{\bar{\mathrm{X}}}$.

For example, the detection condition for the mutation where z is replaced with x in (4.8) is

$$dF^{\mathrm{z}}_{\mathrm{x}} = ((x \leftrightarrow y)w \vee (\overline{vzw})) \oplus ((x \leftrightarrow y)w \vee (\overline{vxw}))$$

It follows that, for instance, a test case $(x, y, z, v, w) = (1, 0, 0, 1, 1)$ will detect the mutation because this assignment of values to variables satisfies $dF^{\mathrm{z}}_{\mathrm{x}}$.

This illustrates a limitation of this definition: the formula is not easy to manipulate, especially when one would like to prove that a property of detection conditions holds for any specification. Proofs in [44] for the restricted case of disjunctive normal form involve manipulating large formulas. Our reformulation and Kuhn's definition are semantically equivalent, but our reformulation allows for more generally applicable, yet more succinct, proofs.

We can compute the *detection condition* as a conjunction of origination condition and propagation condition:

$$dS^{\mathrm{X}}_{\mathrm{E}} = (X \oplus E) \wedge \frac{dS}{dX} \tag{4.9}$$

For example, the detection condition for the mutation where z is replaced with x in (4.8) is

$$dF^{\mathrm{z}}_{\mathrm{x}} = (z \oplus x) \wedge \frac{dF}{dz} = (z \oplus x)(x \oplus y)vw$$

Chapter 5

Analytical Comparison of Mutation Operators

This section uses mutation conditions explained in Section 4 to derive the detectability relationships between several mutation operators. Some practical implications of these results are presented in Section 6.

The notation $\mathcal{S}_{\mathrm{M}}$ is used to represent the detection condition for an arbitrary mutation produced by applying mutation operator M. The detection conditions for mutation operators CRO, CNO, ENO, CCO, and CDO are summarized in Table 5.1. There, x is a clause in S, y is another valid clause[1], and X is an expression in S.

$\mathcal{S}_{\mathrm{CRO}}$	$dS^{\mathrm{x}}_{\mathrm{y}}$	Clause Reference Operator
$\mathcal{S}_{\mathrm{CNO}}$	$dS^{\mathrm{x}}_{\bar{\mathrm{x}}}$	Clause Negation Operator
$\mathcal{S}_{\mathrm{ENO}}$	$dS^{\mathrm{X}}_{\bar{\mathrm{X}}}$	Expression Negation Operator
$\mathcal{S}_{\mathrm{CCO}}$	$dS^{\mathrm{x}}_{\mathrm{x} \wedge \mathrm{y}}$	Clause Conjunction Operator
$\mathcal{S}_{\mathrm{CDO}}$	$dS^{\mathrm{x}}_{\mathrm{x} \vee \mathrm{y}}$	Clause Disjunction Operator

Table 5.1: Detection Conditions for Several Mutation Operators.

First consider the detectability relationship between clause reference operator (CRO) and clause negation operator (CNO).

Theorem 1 *If the clause replaced in $\mathcal{S}_{\mathrm{CRO}}$ is the same clause negated in $\mathcal{S}_{\mathrm{CNO}}$, then $\mathcal{S}_{\mathrm{CRO}} \rightarrow \mathcal{S}_{\mathrm{CNO}}$.* [2]

Proof. By Table 5.1, we must establish that, for a predicate P and a clause x occurring in P, $dP^{\mathrm{x}}_{\mathrm{y}} \rightarrow dP^{\mathrm{x}}_{\bar{\mathrm{x}}}$ holds, where y is another valid clause different from x. Rewriting with (4.9), we have

$$((x \oplus y) \wedge \frac{dP}{dx}) \rightarrow ((x \oplus \bar{x}) \wedge \frac{dP}{dx})$$

Since $x \oplus \bar{x} = 1$, and in view of (4.7), the theorem holds. Q.E.D.

By Theorem 1, a test case, that is, an assignment of values to variables, which makes $dP^{\mathrm{x}}_{\mathrm{y}}$ true, also makes $dP^{\mathrm{x}}_{\bar{\mathrm{x}}}$ true. This is stated as the following corollary.

[1] While Kuhn [44] restricts y to be a variable in S, this thesis only requires $S^{\mathrm{x}}_{\mathrm{y}}$ to be syntactically legal, that is, y has to be a valid clause. This applies to other mutations involving clauses.

[2] Some papers indicate fault class domination by an arrow. In contrast, $\mathcal{S}_{\mathrm{CRO}}$ and $\mathcal{S}_{\mathrm{CNO}}$ are predicates, and the theorem states a logical implication.

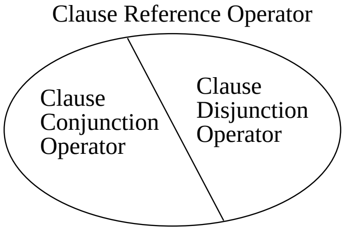

Clause Reference Operator
Clause Conjunction Operator
Clause Disjunction Operator

Figure 5.1: Detectability Relationship between Tests for a Clause Reference Mutation and Tests for the Corresponding Clause Insertion Mutations.

Corollary 1 *Any test case that detects a clause reference mutation for a clause in a predicate will also detect the clause negation mutation for the same clause.*

It must be noted that $dP^{\mathrm{x}}_{\mathrm{y}} \rightarrow dP^{\mathrm{x}}_{\bar{\mathrm{x}}}$ in Theorem 1 does not guarantee the existence of a test for the clause reference mutation. For instance, if $P^{\mathrm{x}}_{\mathrm{y}}$ and P evaluate the same on their entire domain, then no test exists for the mutation, even though there may be a test for the corresponding clause negation mutation. But $dP^{\mathrm{x}}_{\mathrm{y}}$ is universally false in this case, so the theorem is still valid. However, whenever there actually is a test case for the clause reference mutation, that test will detect the clause negation mutation.

Another interesting detectability relationship is between clause negation operator (CNO) and expression negation operator (ENO).

Theorem 2 *If the clause negated in $\mathcal{S}_{\mathrm{CNO}}$ occurs in the expression negated in $\mathcal{S}_{\mathrm{ENO}}$, then $\mathcal{S}_{\mathrm{CNO}} \rightarrow \mathcal{S}_{\mathrm{ENO}}$.*

Proof. By Table 5.1, we must establish that, for a predicate P, with a clause x occurring in a subpredicate E of P, $dP^{\mathrm{x}}_{\bar{\mathrm{x}}} \rightarrow dP^{\mathrm{E}}_{\overline{\mathrm{E}}}$ holds.

Rewriting with (4.9), we have

$$((x \oplus \bar{x}) \wedge \frac{dP}{dx}) \rightarrow ((E \oplus \overline{E}) \wedge \frac{dP}{dE})$$

Since the exclusive-or of a predicate with its negation is trivially true, this can be rewritten as

$$\frac{dP}{dx} \rightarrow \frac{dP}{dE}$$

Since clause x is a subpredicate of E, the theorem follows from Lemma 1. Q.E.D.

Corollary 2 *Any test case that detects a clause negation mutation for a clause in a predicate will also detect an expression negation mutation for an expression in which the clause occurs.*

Consider the detectability relationship between the clause reference operator (CRO) and the two clause insertion operators. Informally, tests for a clause conjunction operator (CCO) mutation and tests for the corresponding clause disjunction operator (CDO) mutation partition the set of test cases that detect the corresponding clause reference mutation. Figure 5.1 presents this relationship. In terms of detection conditions, it means:

1. The disjunction of $\mathcal{S}_{\mathrm{CCO}}$ and $\mathcal{S}_{\mathrm{CDO}}$ is equivalent to $\mathcal{S}_{\mathrm{CRO}}$.

2. $\mathcal{S}_{\mathrm{CCO}}$ and $\mathcal{S}_{\mathrm{CDO}}$ are never satisfied simultaneously.

This is formalized in Theorem 3.

Theorem 3 *If a clause x is replaced with another clause y in $\mathcal{S}_{\text{CRO}}$, and the same clause x is replaced with $x \wedge y$ in $\mathcal{S}_{\text{CCO}}$, and with $x \vee y$ in $\mathcal{S}_{\text{CDO}}$, then*

$$((\mathcal{S}_{\text{CCO}} \vee \mathcal{S}_{\text{CDO}}) \leftrightarrow \mathcal{S}_{\text{CRO}}) \wedge (\overline{\mathcal{S}_{\text{CCO}}} \vee \overline{\mathcal{S}_{\text{CDO}}})$$

Proof. By Table 5.1, we must establish that, for a predicate P and a clause x occurring in P,

$$((dP^{\text{x}}_{\text{x}\wedge\text{y}} \vee dP^{\text{x}}_{\text{x}\vee\text{y}}) \leftrightarrow dP^{\text{x}}_{\text{y}}) \wedge (\overline{dP^{\text{x}}_{\text{x}\wedge\text{y}}} \vee \overline{dP^{\text{x}}_{\text{x}\vee\text{y}}})$$

where y is another valid clause different from x.

By (4.9), the detection conditions for CRO, CCO, and CDO are

$$\begin{aligned} dP^{\text{x}}_{\text{x}\wedge\text{y}} &= (x \oplus (x \wedge y)) \wedge \frac{dP}{dx} = x\bar{y} \wedge \frac{dP}{dx} \quad \text{by (4.2)} \\ dP^{\text{x}}_{\text{x}\vee\text{y}} &= (x \oplus (x \vee y)) \wedge \frac{dP}{dx} = \bar{x}y \wedge \frac{dP}{dx} \quad \text{by (4.4)} \\ dP^{\text{x}}_{\text{y}} &= (x \oplus y) \wedge \frac{dP}{dx} \end{aligned} \tag{5.1}$$

The disjunction of the detection conditions for CCO and CDO is

$$dP^{\text{x}}_{\text{x}\wedge\text{y}} \vee dP^{\text{x}}_{\text{x}\vee\text{y}} = (x\bar{y} \wedge \frac{dP}{dx} \vee \bar{x}y \wedge \frac{dP}{dx}) = (x \oplus y) \wedge \frac{dP}{dx} \tag{5.2}$$

Additionally,

$$\overline{x\bar{y} \wedge \frac{dP}{dx}} \vee \overline{\bar{x}y \wedge \frac{dP}{dx}} = \bar{x} \vee y \vee x \vee \bar{y} \vee \overline{\frac{dP}{dx}} = 1 \tag{5.3}$$

In view of (5.1), (5.2) and (5.3), the Theorem holds. Q.E.D.

Corollary 3 *Any test case that detects a clause insertion mutation in a predicate which replaces a clause x with $x \vee y$ or with $x \wedge y$, y is another valid clause, will also detect the clause reference mutation which replaces the same clause x with y.*

Corollary 4 *Any test case that detects a clause reference mutation in a predicate which replaces a clause x with another valid clause y will also detect either the clause conjunction mutation which replaces the clause x with $x \wedge y$ or the clause disjunction mutation which replaces the clause x with $x \vee y$, but not both.*

Putting the results of this section together, Figure 5.2 depicts the hierarchy of tests that detect various categories of mutation operators in predicates. Note that this hierarchy applies to arbitrary predicates. It is not restricted to predicates in disjunctive normal form.

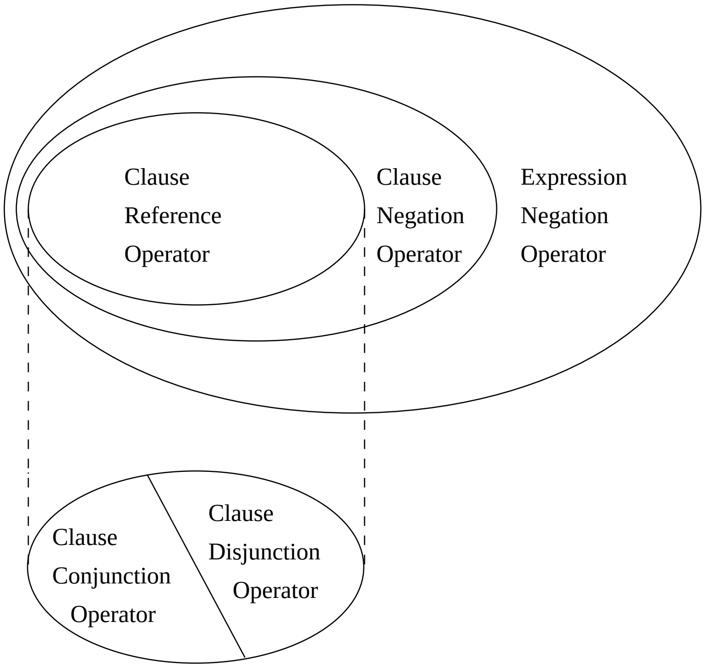


Figure 5.2: Hierarchy of Mutation Operators.

Chapter 6

Applications

The hierarchy in Figure 5.2 is rather general. Why should testing researchers care? This section presents examples of applying the hierarchy and mutation conditions to specific cases. Section 6.2 discusses how the results apply to mutations involving Boolean variables, as well as those involving relational expressions. Section 6.3 explains and discusses previous empirical observations about other mutation operators including logical connector reference and missing clause operators. Section 6.4 analyzes the basic meaningful impact strategy [82].

6.1 Form of Specifications

Kuhn's hierarchy as developed in [44] applies to specifications with formulas in disjunctive normal form. Since actual specifications are generally not in disjunctive normal form, we originally proposed to study to what degree the form of specification affects the tests generated. However, since then we developed the technique that allowed us to prove that Kuhn's hierarchy applies to arbitrary predicates, not just those in disjunctive normal form. Therefore, we conclude that the form of specifications does not affect the applicability of the hierarchy developed in Section 5.

6.2 Comparison of Mutations in Specific Constructs

In this section, we consider application of mutation conditions introduced in Section 4 and the hierarchy developed in Section 5 to mutations in specific constructs: Boolean formulas and relational expressions.

Since the results in Section 5 were proved for a more general case of clauses in predicates, they apply directly to Boolean formulas. These specific applications generalize results in [44, 45] which were limited to specifications in disjunctive normal form.

While fault classes, and therefore mutations, in relational expressions were thoroughly investigated [28, 74], use of mutation conditions allows us to compare the mutation operators from another, more formal, perspective. The clause reference mutation operator includes relational connector reference mutations and off-by-1 mutations. By Corollary 1,

- Any test case that detects a relational connector reference mutation for a clause in a predicate will also detect a clause negation mutation for the same clause.

- Any test case that detects an off-by-1 mutation for a clause in a predicate will also detect a clause negation mutation for the same clause.

For a relational expression, there are four possible relational connector reference mutations for each of the other relational connectors, as well as two possible off-by-1 mutations. Consider, for instance, a specification S with a clause $E < F$, where E and F are arithmetic expressions.

The four relational connector reference mutations replace $E < F$ with $E \leq F$, $E > F$, $E \neq F$, and $E = F$. [1] The two off-by-1 mutations replace $E < F$ with $E < F + 1$ and $E < F - 1$. Mutation conditions can be used to compare these mutations. The detection condition for an off-by-1 mutation which replaces $E < F$ with $E < F - 1$ is

$$\begin{aligned} dS^{\mathrm{E<F}}_{\mathrm{E<F-1}} &= ((E < F) \oplus (E < F - 1)) \wedge \frac{dS}{d(E < F)} \\ &= (E = F - 1) \wedge \frac{dS}{d(E < F)} \end{aligned}$$

since $E < F - 1$ evaluates to a different value than $E < F$ only when E is equal to $F - 1$. On the other hand, the detection condition for a relational connector reference mutation that replaces $E < F$ with $E > F$ is

$$dS^{\mathrm{E<F}}_{\mathrm{E>F}} = ((E < F) \oplus (E > F)) \wedge \frac{dS}{d(E < F)}$$

Since $E < F$ and $E > F$ are never satisfied simultaneously,

$$\begin{aligned} dS^{\mathrm{E<F}}_{\mathrm{E>F}} &= ((E < F) \vee (E > F)) \wedge \frac{dS}{d(E < F)} \\ &= (E \neq F) \wedge \frac{dS}{d(E < F)} \end{aligned}$$

Since $(E = F - 1) \rightarrow (E \neq F)$, $dS^{\mathrm{E<F}}_{\mathrm{E<F-1}} \rightarrow dS^{\mathrm{E<F}}_{\mathrm{E>F}}$.

It follows that any test case that detects an off-by-1 mutation which replaces a clause $E < F$ with $E < F - 1$ will also detect a relational connector mutation which replaces the same clause with $E > F$.

In a similar fashion, it is possible to derive a set of detectability relationships between relational connector and off-by-1 mutations for various relational expressions.

6.3 Analysis of Previous Observations

Gopal and Budd [32] note that a logical connector reference operator (LRO) mutation, where $\vee$ is substituted for $\wedge$ and vice versa, tends to be trivial to detect. Indeed, in view of (4.6), the detection condition for such a mutation in a specification $S = P \vee Q$ is

$$dS^{\mathrm{P\vee Q}}_{\mathrm{P\wedge Q}} = (P \vee Q) \oplus (P \wedge Q) = P \oplus Q,$$

so this mutation is detected by any test where P and Q evaluate differently. This explains Gopal and Budd's observation.

The detection condition for a corresponding missing expression operator (MEO) mutation is

$$dS^{\mathrm{P\vee Q}}_{\mathrm{P}} = (P \vee Q) \oplus P = \bar{P} \wedge Q,$$

and $dS^{\mathrm{P\vee Q}}_{\mathrm{P}} \rightarrow dS^{\mathrm{P\vee Q}}_{\mathrm{P\wedge Q}}$. Hence, a test that detects an MEO mutation for an operand of an $\vee$ connector will also detect an LRO mutation where the connector is replaced with $\wedge$. A similar result can be obtained when $\wedge$ is replaced with $\vee$.

The above does not mean that all LRO mutants are easy to detect. In Section 3.2.2, we introduced LRO$_2$ mutation operator which replaces $\vee$ with $\oplus$, $\wedge$ with $\leftrightarrow$, $\rightarrow$ with $\leftrightarrow$. Consider an LRO$_2$ mutation where $\vee$ is replaced with $\oplus$. The detection condition is

$$dS^{\mathrm{P\vee Q}}_{\mathrm{P\oplus Q}} = (P \vee Q) \oplus (P \oplus Q) = P \wedge Q,$$

[1] By definition of the relational connector reference operator in Section 3.2.1, we do not replace $E < F$ with $E \geq F$, since that is the same as negating the expression.

so the mutation is relatively hard to detect. This is reasonable since $\vee$ differs from $\oplus$ in only one of four positions of the truth table, while it differs from $\wedge$ in two positions. Similarly, the other LRO_2 mutations are difficult to detect. We evaluate the effectiveness of combining mutation operators with LRO_2 in Section 7.4.

Stuck-at operator (STO) is one of the suboperators of the clause reference operator. Therefore, any test that detects a stuck-at mutation for a clause in a predicate will also detect a clause negation mutation for the same clause. It was suggested in [44] that missing clause operator (MCO) mutation can be regarded as a special case of variable reference mutation. However, it is more appropriate to compare MCO with STO. For instance, if a specification contains a conjunction $x \wedge y$, an MCO mutant where clause y is omitted is equivalent to an STO mutant where y is replaced with 1. Similarly, in $x \vee y$ or $x \oplus y$, an MCO mutant for clause y is equivalent to an STO mutant where y is replaced with 0. Implication is a special case. Consider specification S containing an implication $x \rightarrow y$. There are two cases. First, an MCO mutant where clause x is omitted is equivalent to an STO mutant where x is replaced with 0. Second, the detection condition for an MCO mutant where y is omitted is

$$dS_{\mathrm{x}}^{\mathrm{x}\rightarrow\mathrm{y}} = ((x \rightarrow y) \oplus x) \wedge \frac{dS}{dy} = (\bar{x} \vee \bar{y}) \wedge \frac{dS}{dy}.$$

On the other hand, the detection condition for an STO mutation where y is replaced with 1 is

$$dS_{1}^{\mathrm{y}} = ((x \rightarrow y) \oplus (x \rightarrow 1)) \wedge \frac{dS}{dy} = x\bar{y} \wedge \frac{dS}{dy}.$$

Since $x\bar{y} \rightarrow (\bar{x} \vee \bar{y})$, $dS_{1}^{\mathrm{y}} \rightarrow dS_{\mathrm{x}}^{\mathrm{x}\rightarrow\mathrm{y}}$.

To summarize, if a test generation strategy guarantees detection of both stuck-at-0 and stuck-at-1 mutations for a clause, it will also guarantee detection of the missing clause mutation for the same clause.

6.4 On the Basic Meaningful Impact Strategy

In this section, we use mutation conditions to show that the basic meaningful impact strategy [82] is stronger in that it tests for stuck-at mutations and not variable negation mutations as proposed by the authors. Weyuker et al. [82] designed the *meaningful impact strategy* for testing Boolean formulas in *irreducible* disjunctive normal form. A formula is said to be in irreducible disjunctive normal form when none of the formula's literals or terms can be deleted without altering the formula's value for some test case.

We briefly repeat here the relevant definitions. More details are in [82]. Let F be a Boolean formula in irreducible disjunctive normal form with n variables and m product terms: $p_1 \vee p_2 \vee \ldots \vee p_m$. Each term is a conjunction of literals. Recall that a literal is a single occurrence of a variable or its negation.

The points of the input space are divided into two categories: *true points* and *false points* are those sets of inputs that cause the formula to evaluate to 1 and 0, respectively. True points for the term p_i are the points of the input space that cause p_i to evaluate to 1. The *unique true points* for the term p_i are the true points for p_i which are not true points for any other term p_j. Denote the set of unique true points for p_i by U_i.

Let $p_{i,j}$ denote the product-term obtained by complementing the jth literal of the product-term p_i. Denote the set of true points for $p_{i,j}$ by $D_{i,j}$. Denote the points in $D_{i,j}$ that are false points for F by $N_{i,j}$.

The basic meaningful impact strategy is defined as follows [82]:

1. Select one test point from each nonempty U_i of F.

2. Select one test point from each $N_{i,j}$ of F.

Weyuker et al. [82] claim that the strategy is testing directly for variable negation mutations. In fact, the strategy is stronger: it is testing for stuck-at mutations.

To show this, we first compute the propagation conditions for a mutation in an arbitrary term and for a mutation in an arbitrary literal for a specification in disjunctive normal form. Since F can be rewritten as

$$p_i \vee (p_1 \vee \ldots \vee p_{i-1} \vee p_{i+1} \ldots \vee p_m),$$

it follows that the propagation condition for a mutation in p_i is

$$\begin{aligned} \frac{dF}{dp_i} &= \overline{p_1 \vee \ldots p_{i-1} \vee p_{i+1} \vee \ldots p_m} \\ &= \overline{p_1} \wedge \ldots \overline{p_{i-1}} \wedge \overline{p_{i+1}} \wedge \ldots \overline{p_m} \end{aligned} \tag{6.1}$$

Let $p_i = l_1 \ldots l_k$, where l_j denotes the jth literal in p_i. Then F can be rewritten as

$$l_j l_1 \ldots l_{j-1} l_{j+1} \ldots l_k \vee (p_1 \vee \ldots \vee p_{i-1} \vee p_{i+1} \ldots \vee p_m)$$

The propagation condition for a mutation in l_j is

$$\frac{dF}{dl_j} = l_1 \ldots l_{j-1} l_{j+1} \ldots l_k \wedge \frac{dF}{dp_i} \tag{6.2}$$

In view of (6.1), the detection condition for a stuck-at-0 mutation which replaces any literal in the term p_i with 0 is

$$(p_i \oplus 0) \wedge \frac{dF}{dp_i} = p_i \wedge \overline{p_1} \wedge \ldots \overline{p_{i-1}} \wedge \overline{p_{i+1}} \wedge \ldots \overline{p_m}$$

This defines the set U_i of unique true points for the term p_i.

In view of (6.2), the detection condition for a stuck-at-1 mutation which replaces literal l_j in the term p_i with 1 is

$$(l_j \oplus 1) \wedge \frac{dF}{dl_j} = \overline{l_j} l_1 \cdots l_{j-1} l_{j+1} \cdots l_k \wedge \frac{dF}{dp_i}$$

This defines the set $N_{i,j}$, since $\overline{l_j} l_1 \cdots l_{j-1} l_{j+1} \cdots l_k$ defines the set $D_{i,j}$ of true points for $p_{i,j}$.

The basic meaningful impact strategy happens to also detect the variable negation mutations because, as we observed in Section 6.3, test cases that detect stuck-at mutations will also detect variable negation mutations.

6.5 Clause Insertion Operator and Pairwise Testing

In this section we theoretically compare clause insertion mutation operator (CIO) and pairwise testing for Boolean specifications. We show that a test case that detects a CIO mutation involving a pair of variables will also cover a pair of value assignments for that pair of variables.

CIO replaces a clause a with $a \wedge b$, $a \vee b$, $a \wedge \bar{b}$, and $a \vee \bar{b}$ for every other clause b. In general, this could result in a huge number of mutations, so the applicability of this operator is limited. However, the mutation operator is clearly defined for Boolean specifications.

Consider a pair of variables x and y. Consider a clause conjunction operator (CCO) mutation which replaces an occurrence of x with $x \wedge y$. By Table 5.1, the detection condition for this mutation is

$$dS^{\mathrm{x}}_{\mathrm{x \wedge y}} = (x \oplus (x \wedge y))\frac{dS}{dx} = x\bar{y}\frac{dS}{dx}$$

It follows that

$$dS^{\mathrm{x}}_{\mathrm{x \wedge y}} \rightarrow x\bar{y}, \tag{6.3}$$

so a test that detects $dS^{\mathrm{x}}_{\mathrm{x \wedge y}}$ will also cover a pair $(x.1, y.0)$ [2].

Similarly, consider a clause disjunction operator (CDO) mutation which replaces an occurrence of y with $y \vee x$. By Table 5.1, the detection condition for this mutation is

$$dS^{\mathrm{y}}_{\mathrm{y \vee x}} = (y \oplus (y \vee x))\frac{dS}{dy} = \bar{y}x\frac{dS}{dy}$$

It follows that

$$dS^{\mathrm{y}}_{\mathrm{y \vee x}} \rightarrow x\bar{y},$$

so a test that detects $dS^{\mathrm{y}}_{\mathrm{y \vee x}}$ will also cover a pair $(x.1, y.0)$. Similar results can be shown for other possible value pairs. We summarize them in Theorem 4.

Theorem 4 *In a Boolean specification, let x and y be two different variables.*

1. *Any test that detects a CCO mutation where an occurrence of x is replaced with $x \wedge y$ or a CDO mutation where an occurrence of y is replaced with $x \vee y$, will also cover a pair $(x.1, y.0)$.*
2. *Any test that detects a CCO mutation where an occurrence of x is replaced with $x \wedge \bar{y}$ or a CCO mutation where an occurrence of y is replaced with $\bar{x} \wedge y$, will also cover a pair $(x.1, y.1)$.*
3. *Any test that detects a CDO mutation where an occurrence of x is replaced with $x \vee \bar{y}$ or a CDO mutation where an occurrence of y is replaced with $\bar{x} \vee y$, will also cover a pair $(x.0, y.0)$.*

We must note that, for instance, (6.3) does not guarantee the existence of a test for the clause conjunction mutation. If the detection condition for the mutation is universally false, then no test exists for the mutation. Our experiments, detailed in Section 7.5, show that clause insertion mutation operator achieves very high pairwise coverage.

6.6 Limitations of Theoretical Comparison

The hierarchical detectability relationship between mutation operators does not always guarantee that the harder-to-detect operator is more effective than the easier-to-detect operator. The following describes such situations.

- A mutation operator generates no mutants.

  Consider operand reference operator (ORO) and clause negation operator (CNO). Since ORO is a suboperator of clause reference operator (CRO), it follows from Corollary 1 that ORO is harder to detect than CNO. However, if a system has only one variable, operand reference operator (ORO) does not generate any mutants, whereas clause negation operator (CNO) generates a mutant where the variable occurrence is negated. This mutant is likely to produce a test case, whereas ORO produces no test cases. This is unlikely to happen in specifications of reasonable size.

- No test exists for a mutant.

  Again, consider ORO and CNO mutation operators. Consider a specification $S = a(a \leftrightarrow b)$. The detection condition for a CNO mutant where the first occurrence of a is negated is $dS^{\mathrm{a}}_{\bar{\mathrm{a}}} = (a \leftrightarrow b)$, and a possible test is $(a, b) = (1, 1)$. However, the detection condition for an ORO mutant where the first occurrence of a is replaced with b is $dS^{\mathrm{a}}_{\mathrm{b}} = (a \oplus b)(a \leftrightarrow b) = 0$,

[2] This notation is explained in Section 2.1.3.

in other words, the detection condition is universally false, so this mutant is equivalent to the original and does not produce any test.

As we mentioned in Section 5, the hierarchical detectability relationship between mutation operators does not guarantee the existence of a test for a harder-to-detect mutation. It only guarantees that if there is a test case for a harder-to-detect mutation, it will detect the corresponding easier-to-detect mutation.

We can augment the harder-to-detect mutation operator in the following way. First apply the harder-to-detect mutation operator and produce tests for those mutants. Whenever a harder-to-detect mutant is equivalent to the original, generate the corresponding mutant from the easier-to-detect mutation operator in the hierarchy and produce a test for that mutant. This approach was suggested by Lau and Yu [45]. It can guarantee detection of the mutants from the easier-to-detect mutation operator.

Note that the specification $S = a(a \leftrightarrow b)$, as well as other examples that we were able to come up with, is rather unnatural: it is equivalent to ab. Since the same test is usually derived from a number of mutants, in practice it is unlikely that a test set that detects all mutants from the harder-to-detect mutation operator does not detect mutants from the easier-to-detect operator.

- An easier-to-detect operator produces more tests.

  Consider CNO and expression negation operator (ENO). By corollary 2, CNO is harder to detect than ENO. ENO generates all the mutants that CNO does, but also additional mutants. Consider a specification $S = a \vee b$. There are two CNO mutants. The detection condition for a CNO mutant where a is negated is

  $$dS^{\mathrm{a}}_{\bar{\mathrm{a}}} = (a \vee b) \oplus (\bar{a} \vee b) = \bar{b},$$

  and a possible test is $(a, b) = (1, 0)$. Similarly,

  $$dS^{\mathrm{b}}_{\bar{\mathrm{b}}} = (a \vee b) \oplus (a \vee \bar{b}) = \bar{a},$$

  and a possible test is $(a, b) = (0, 1)$. So CNO produces two tests. ENO generates the two CNO mutants and an additional mutant where the whole expression is replaced. The detection condition for this mutant is universally true, any test will satisfy it. The test generation process may choose to produce a test which is different from the two tests produced by the CNO mutants, so that ENO produces three tests. While the test set produced by ENO is not more effective at detecting the expression negation mutants than the test set produced by CNO, it may be more effective at detecting mutations produced by other mutation operators.

  The degree of difference in effectiveness can be studied empirically. We show in Section 7 that CNO is almost as effective in practice as ENO while generating far fewer mutants.

For some mutation operators, we can prove that whenever a mutant from an easier-to-detect mutation operator produces a test, the corresponding mutant from a harder-to-detect operator is guaranteed to produce a test. Consider stuck-at-operator (STO) and CNO. STO is a suboperator of CRO, so by Corollary 1, STO is harder to detect than CNO. The detection condition for stuck-at-0 mutation which replaces clause x with 0 in specification S is

$$dS^{\mathrm{x}}_{0} = (x \oplus 0) \wedge \frac{dS}{dx} = x\frac{dS}{dx}.$$

Similarly, the detection condition for the corresponding stuck-at-1 mutation is

$$dS^{\mathrm{x}}_{1} = \bar{x}\frac{dS}{dx}.$$

Assume that there is a test for the corresponding CNO, that is,

$$dS^{\mathrm{x}}_{\bar{\mathrm{x}}} = \frac{dS}{dx}$$

is not universally false. Since at least one of x and $\bar{x}$ will satisfy $\frac{dS}{dx}$, at least one of dS^{x}_0 and dS^{x}_1 is not universally false. Therefore, there is a test for either stuck-at-0 or stuck-at-1 fault. This implies that the tests for STO are adequate for detecting CNO mutants.

Chapter 7

Empirical Comparison of Mutation Operators

Aside from the special situations listed in Section 6.6, the theoretical comparison does not provide a quantitative measure of the differences between the mutation operators. Additionally, the detection conditions for some mutation operators are incomparable. In this section, we empirically compare the effectiveness and cost of mutation operators and sets of operators by evaluating their coverage and the number of mutants they generate.

We developed a 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 tool is described in Appendix A.

We now describe the experimental base and also provide some details of experiment setup.

7.1 Cruise Control

Many variations of the automobile cruise control specification exist in the literature [41, 5, 4]. We use the specification from [5]. This version of the specification does not model throttle and has four modes: Off - ignition is off, Inactive - ignition is on, cruise control system is off, Cruise - cruise control system is controlling the speed, Override - cruise control system is on but not controlling the speed. The system starts in Off.

The system's environmental conditions indicate whether the automobile's ignition is on (*Ignited*), the engine is running (*EngRun*), the automobile is going too fast to be controlled (*Toofast*), the brake pedal is being pressed (*Brake*), and whether the cruise control level is set to *Activate*, *Deactivate*, or *Resume*.

Each row in Table 7.1 specifies a conditioned event that activates a transition from the mode on the left to the mode on the right. A table entry of @T or @F under a column header C represents a triggering event @T(C) or @F(C). "@T(C)" means C must change from false to true for the transition to be taken, and "@F(C)" means C must change from true to false. A table entry of `t` under a column header C means the transition can only be taken if C is true. Similarly, an entry of `f` means it can only be taken if C is false. If the value of a condition C does not affect a conditioned event, the table entry is marked with a hyphen "-" ("don't care" condition).

The SMV specification of cruise control was derived in [3]. For example, row 8 of Table 7.1 states that if cruise control is in mode Cruise, when Ignited is true, EngRun is true, Toofast is false, and if Deactivate changes from false to true, cruise control will change into mode Override. In the SMV transition model, this is represented as follows:

Previous Mode	Igni-ted	Eng-Run	Too-fast	Brake	Acti-vate	Deac-tivate	Resume	New Mode
Off	@T	-	-	-	-	-	-	Inactive
Inactive	@F	-	-	-	-	-	-	Off
	t	t	-	f	@T	-	-	Cruise
Cruise	@F	-	-	-	-	-	-	Off
	t	@F	-	-	-	-	-	Inactive
	t	-	@T	-	-	-	-	
	t	t	f	@T	-	-	-	Override
	t	t	f	-	-	@T	-	
Override	@F	-	-	-	-	-	-	Off
	t	@F	-	-	-	-	-	Inactive
	t	t	-	f	@T	-	-	Cruise
	t	t	-	f	-	-	@T	

Table 7.1: SCR Specifications for the Cruise Control System.

```
next(CruiseControl) := case
    ...
    CruiseControl=Cruise  & Ignited & EngRun & !Toofast &
    !(Enum1=Deactivate) & next(Enum1)=Deactivate : Override;
    ...
esac;
```

Since the cruise control level can be set to only one value at a time, the specification has a scalar variable Enum1 with domain: {Activate, Deactivate, Resume}. The guard in the above guarded command is reflected into temporal logic as follows:

```
PCruiseControl=Cruise  & PIgnited & PEngRun & !PToofast &
    !(PEnum1=Deactivate) & Enum1=Deactivate
```

Ammann et. al [3] introduced extra variables, such as PCruiseControl, Pignited, representing the values from the previous state, in order to reflect the triggering events in temporal logic. Recall that a triggering event specifies two values: a before-value and an after-value.

7.1.1 Reflection Details

The case structure for CruiseControl variable has 12 guarded commands, one for each row of Table 7.1. We start by making the targets of these commands pairwise disjoint. This involves combining the guarded commands which have the same targets. Since CruiseControl has four modes, we get four guarded commands. We then expound to recast the guards to be a partition. Recall that expoundment is a process that makes implicit parts of a specification explicit.

Consider the transitions that lead to mode Off. Rows 2, 4, and 9 of Table 7.1 explicitly describe the transitions from the other 3 modes to mode Off. All of the three transitions have the same triggering event: changing the value of Ignited to False, the transitions do not depend on the value of other environmental conditions. For example, the guard for transition from Override to Off is

```
PCruiseControl=Override & PIgnited & !Ignited
```

There is also an implicit transition from Off to itself. CruiseControl remains in mode Off unless the value of Ignited changes to True. After combining the guards by OR-ing them, then simplifying, we get the following formula for guard1:

```
PIgnited & !Ignited | PCruiseControl=Off & (PIgnited | !Ignited)
```

Then the reflected formula to be mutated is

```
SPEC AG AX (guard1 <-> (PIgnited & !Ignited |
    PCruiseControl=Off & (PIgnited | !Ignited)))
```

The next step temporal operator (AX) is needed to avoid generating counterexamples which pinpoint an inconsistency in the initial step where the before-values for variables are meaningless. We then apply mutation operators to the right hand side of the equivalence connector. The guards for transitions to the other three modes are derived similarly. Note that we do not modify the state machine itself.

7.2 Other Specifications

Cruise control lacks certain features which we would like to study. To broaden the experimental base, we choose four additional specifications with the following features:

- Three specifications have relational expressions.
- Trusted OS is a large specification.
- TCAS/Siemens has an internal state which is large relative to the number of inputs and outputs.
- TCAS II/Boolean is a set of Boolean specifications which was studied extensively in testing research.

For each sample specification, we apply guard reflection whenever possible. We also apply direct reflection to the cases where guard reflection is not possible or useful, such as DEFINE declarations which do not contain *case* statements. Guard reflection and direct reflection are explained in Section 2.6.1.

Safety Injection

Safety injection [7] describes a part of a nuclear reactor safety system. If the water pressure is too low, extra water is injected, unless overridden. The system is overridden depending on the pressure, whether the override is blocked, and whether it is reset. The SMV specification of safety injection is given in [3].

TCAS/Siemens

TCAS, aircraft collision avoidance, is a part of a set of C programs that came originally from Siemens Corporate Research [40] and was subsequently modified by Rothermel and Harrold [72]. These programs are used in research on program testing, so they come with extensive test suites and sets of faulty versions. There are 12 input variables specifying parameters of own aircraft and another aircraft and one output variable, `alt_sep`, a resolution advisory to maintain safe altitude separation between the two aircrafts. The program computes intermediate values and prints `alt_sep` to the standard output.

The program has minimal documentation, and we wrote a formal specification for it.

Trusted OS

This is a portion of a trusted operating system [87]. The simplified model consists of one process and one file. Files have an owner, a group, and a security level. Files also have permission bits to allow a file to be read or written by the owner, anyone in the group, and others. Processes have a user and group, and may have privileges to raise or lower the security level of a file. The experiment concentrated on testing a command for changing the security level of a file.

ORO	Operand Reference Operator
STO	Stuck-At Operator
MEO	Missing Expression Operator
MCO	Missing Clause Operator
LRO	Logical Connector Reference Operator
ENO	Expression Negation Operator
CNO	Clause Negation Operator
ASO	Associative Shift Operator
OFO	Off-By-1 Operator
RRO	Relational Connector Reference Operator

Table 7.2: Mutation Operators.

	Mutants	U - I Mutants	Unique Traces
Cruise Control	1104	384	51
Safety Injection	615	135	22
TCAS/Siemens	1020	448	82
Trusted OS	7224	2276	422

Table 7.3: Number of Mutants and Traces for the State-based Specifications.

TCAS II/Boolean

Weyuker et. al [82] selected 20 transition specifications from the specification of TCAS II, an aircraft collision avoidance system described by Leveson et. al [46]. Figure B.1 in Appendix B lists these Boolean specifications. We chose to include these specifications in our study although they are not state-based specifications. Boolean specifications are often used to specify complex systems, and TCAS II/Boolean has been studied by many researchers [82, 16, 42].

7.3 Evaluation of Mutation Operators

We ran experiments on the sample SMV specifications to compare the mutation operators in terms of the number of mutants and test cases produced and the specification coverage. We list mutation operators in Table 7.2. These operators, as well as LRO_1 and LRO_2, are defined in Section 3.2.

Table 7.3 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 all mutation operators to the sample specifications. Since CNO mutants are a subset of ENO mutants, and MCO mutants are a subset of MEO mutants, we do not include CNO and MCO mutants in the total number of mutants.

We use the specification-based coverage metric introduced in [2]. 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 semantically unique, inconsistent (U-I) mutants generated by all operators for a given example (the number of U-I mutants is much smaller than the total number of mutants, see Table 7.3). 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 $\frac{k}{N}$.

We present details of experiments in Tables 7.4, 7.5, 7.6, and 7.7. As in Table 7.3, “Mutants” is the total number of mutants generated by each operator, including consistent and duplicate mutants. “UTs” is the number of unique traces generated by SMV after duplicate traces and prefixes are removed.

Operator	Mutants	UTs	Coverage
ORO	442	34	95.8
STO	148	29	96.1
MEO	132	33	96.9
MCO	70	26	94.0
LRO	198	32	94.8
LRO_1	132	24	92.7
LRO_2	66	18	78.4
ENO	140	20	82.8
CNO	74	16	82.0
ASO	44	19	90.1

Table 7.4: Cruise Control Scores of Mutation Operators.

Operator	Mutants	UTs	Coverage
ORO	469	73	97.8
STO	134	28	92.9
MEO	80	24	91.5
MCO	52	21	87.3
LRO	120	23	90.2
LRO_1	80	14	83.5
LRO_2	40	14	77.5
ENO	116	19	82.1
CNO	67	19	82.1
ASO	11	5	54.7
OFO	26	9	56.2
RRO	56	18	64.7

Table 7.5: TCAS/Siemens Scores of Mutation Operators.

Results for RRO and OFO do not appear for Cruise Control, since it does not have any relational expressions.

Figure 7.1 shows mutation coverage plotted against percentage of total mutants. The data points represent averages for the four state-based specifications. Figure 7.1(a) presents the results for all mutation operators, while Figure 7.1(b) provides a magnified view of the results for the mutation operators with average mutation coverage ranging from 78.03% for LRO_2 to 90.88% for STO. "Utopia" marks the best outcome: 100% coverage at no cost. All plots in this thesis were generated using Dataplot [27].

Discussion

As shown in Figure 7.1, operand reference operator (ORO) generates by far the largest number of mutants, but provides the best coverage of any single operator. Stuck-at operator (STO), missing expression operator (MEO), and logical connector reference operator (LRO) provide second best coverage while generating far fewer mutants. Missing clause operator (MCO) provides less coverage while generating even fewer mutants. Clause negation operator (CNO) and expression negation operator (ENO) get less coverage than MCO, while ENO generates many more mutants. Several other operators generate very few mutants, and their coverage is low. We consider LRO_2 further in Section 7.4.

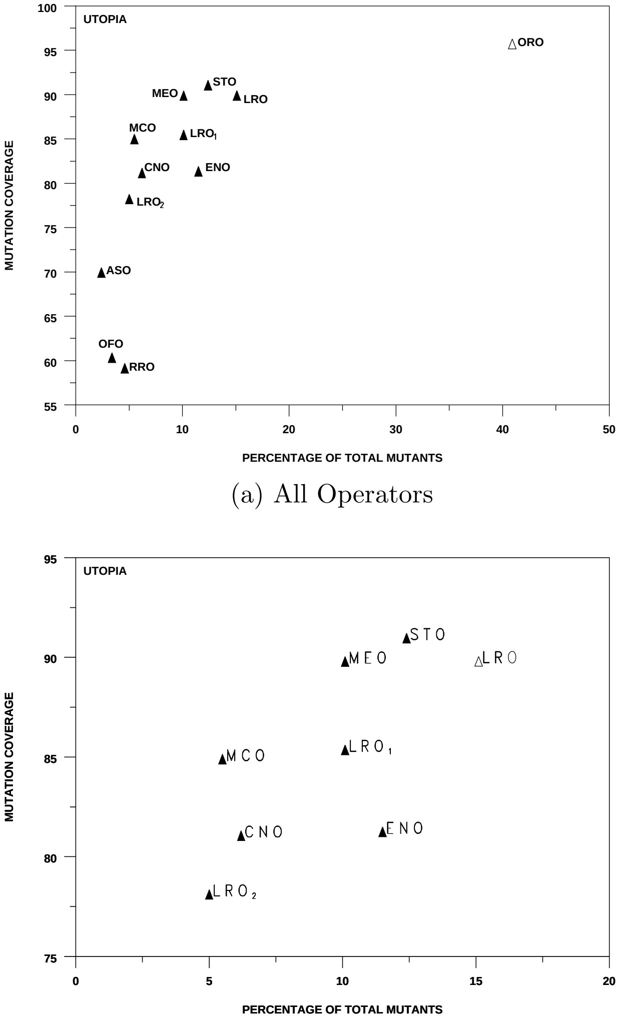
To explain why ORO generates far more mutants than any other single operator, we note that if the number of atoms (variables and constants) in a specification is V and the number of value references is R, ORO results in $O(V * R)$ mutants, whereas CNO, LRO, MCO, STO, ASO

Operator	Mutants	UTs	Coverage
ORO	140	11	90.4
STO	82	9	88.9
MEO	74	9	88.9
MCO	37	6	81.5
LRO	111	9	88.9
LRO_1	74	6	83.0
LRO_2	37	5	83.0
ENO	78	7	83.7
CNO	41	7	83.7
ASO	22	5	80.7
OFO	36	7	67.4
RRO	72	8	66.7

Table 7.6: Safety Injection Scores of Mutation Operators.

Operator	Mutants	UTs	Coverage
ORO	3949	373	98.2
STO	696	94	85.7
MEO	614	71	81.5
MCO	322	55	76.5
LRO	921	114	85.0
LRO_1	614	71	82.0
LRO_2	307	74	73.3
ENO	671	59	76.0
CNO	348	59	76.0
ASO	81	19	53.5
OFO	134	24	56.8
RRO	88	16	45.5

Table 7.7: Trusted OS Scores of Mutation Operators.



(a) All Operators

(b) Some Operators (Magnified View)

Figure 7.1: Cost-effectiveness of Mutation Operators (Average for the State-based Specifications).

and RRO result in $O(R)$ mutants.

Cruise control has an unusually high coverage for ASO. The reason is that the reflected CTL formulas are relatively long and contain a large number of logical connectors and parenthesis, so that there are many ASO mutants. ASO often generates mutants which are hard to detect; however, for most specifications it generates very few mutants.

In [10], STO, CNO, and ENO had similar coverage. The reason for this discrepancy is the use of direct reflection in [10], as opposed to guard reflection in this study. Direct reflection tends to disfavor STO. Consider the following CTL formula produced by direct reflection:

```
AG (x & y -> AX (w = 1))
```

Stuck-at-0 mutation of x or y will result in a mutant which is always consistent with a state machine. In general, only the mutants where the left hand side of implication is true may produce counterexamples. Guard reflection does not have this limitation.

As we explained in Section 6.6, the hierarchical detectability relationship between mutation operators does not always guarantee that the harder-to-detect operator is more effective than the easier-to-detect operator. For instance, while CNO is harder to detect than ENO, the latter generates more mutants and therefore may produce produce more tests. However, the experimental results show that this has only minimal impact. In practice, there are very few cases where CNO cannot detect a mutant which ENO can detect.

7.4 Evaluation of Mutation Operator Sets

We are interested in finding sets of operators which improve upon the coverage of corresponding individual operators without generating too many mutants.

As we showed in Section 6.3, an LRO_2 mutation is difficult to detect. (Recall that LRO_2 replaces $\vee$ with $\oplus$, $\wedge$ with $\leftrightarrow$, $\rightarrow$ with $\leftrightarrow$.) Truth tables provide an intuitive explanation for such a selection. Consider a specification $S = P \vee Q$. Table 7.8 presents its truth values in terms of P and Q; it also gives the truth values for 4 functions each differing from S in exactly one truth value. The last 3 columns could be produced by several semantically close mutants. For example, stuck-at-0 operator which replaces Q with 0 produces the same mutant as missing expression operator which omits Q. However, it is hard to think of any other mutation operator that produces the same truth values as $P \oplus Q$. This uniqueness suggests that LRO_2 will be a useful addition to other mutation operators.

P	Q	$P \vee Q$	$P \oplus Q$	P	Q	$1 \vee Q$
1	1	1	$\underline{\mathbf{0}}$	1	1	1
0	1	1	1	$\underline{\mathbf{0}}$	1	1
1	0	1	1	1	$\underline{\mathbf{0}}$	1
0	0	0	0	0	0	$\underline{\mathbf{1}}$

Table 7.8: Truth values for $P \vee Q$ and its Close Mutants.

To compare the combinations of mutation operators, we evaluate the specification-based coverage of resulting test sets using the mutation coverage metric. We present details of experiments in Tables 7.9, 7.10, 7.11, and 7.12. The column names are the same as in corresponding Tables for mutation operators in Section 7.3. LRO already includes LRO_2 operator so its results are unchanged from Section 7.3.

Figure 7.2 shows mutation coverage of each mutation operator set plotted against its percentage of total mutants. The data points represent averages for the four state-based specifications.

In Table 7.13, we present average mutation coverage of individual mutation operators and mutation operator sets for these specifications. Coverage of ORL^{+}, which is not shown in Table 7.13, is 99.5%. Additionally, Figure 7.3 visually compares the average mutation coverage of

Operator Set	Mutants	UTs	Coverage
ORL^+	508	45	98.4
ORL	508	45	98.4
STL	214	42	98.4
MEL	198	42	98.4
MCL	136	39	96.6
LRO	198	32	94.8
ENL	206	27	92.5
CNL	140	27	92.5
ASL	110	28	93.8
OFL	66	18	78.4
RRL	66	18	78.4

Table 7.9: Cruise Control Scores of Mutation Operator Sets.

Operator Set	Mutants	UTs	Coverage
ORL^+	591	81	99.8
ORL	509	78	98.9
STL	174	36	95.1
MEL	120	32	94.0
MCL	92	30	92.0
LRO	120	23	90.2
ENL	156	28	92.0
CNL	107	28	92.0
ASL	51	17	80.1
OFL	66	20	79.7
RRL	96	26	82.4

Table 7.10: TCAS/Siemens Scores of Mutation Operator Sets.

Operator Set	Mutants	UTs	Coverage
ORL^+	285	22	100.0
ORL	177	14	91.8
STL	119	11	90.4
MEL	111	11	90.4
MCL	74	10	88.2
LRO	111	9	88.9
ENL	115	9	88.9
CNL	78	9	88.9
ASL	59	9	89.6
OFL	73	10	88.9
RRL	109	11	88.2

Table 7.11: Safety Injection Scores of Mutation Operator Sets.

Operator Set	Mutants	UTs	Coverage
ORL$^+$	4478	419	99.9
ORL	4256	417	99.9
STL	1003	139	88.9
MEL	921	119	85.8
MCL	629	109	82.8
LRO	921	114	85.0
ENL	978	111	85.0
CNL	655	111	85.0
ASL	388	85	77.1
OFL	441	87	76.5
RRL	395	83	74.2

Table 7.12: Trusted OS Scores of Mutation Operator Sets.

the mutation operator sets and their corresponding individual mutation operators. For example, it shows, side-by-side, coverage of the mutation operator set RRL and its corresponding individual operator RRO. Note that ORO is an individual operator that corresponds to both ORL and ORL$^+$.

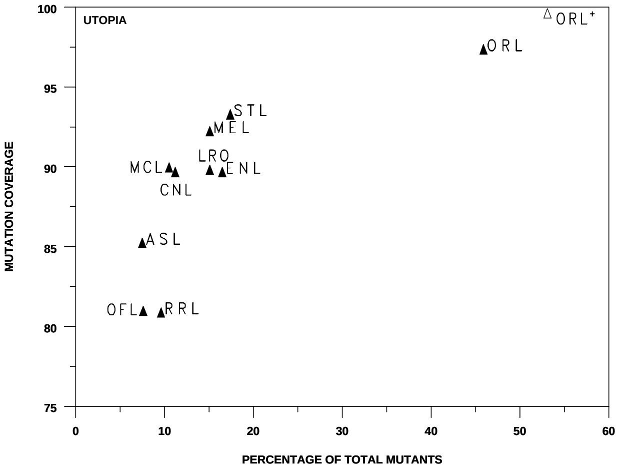


Figure 7.2: Cost-effectiveness of Mutation Operator Sets (Average for the State-based Specifications).

We also evaluated mutation coverage for the set of 20 specifications from TCAS II/Boolean. In the case of Boolean specification, the action of mutation operators is simpler, for example, ORO replaces a variable occurrence with another variable. In Table 7.14, we present average mutation coverage of individual mutation operators and mutation operator sets for TCAS II/Boolean specifications. The details for each individual specification as well as the averages can be found in Appendix B.

Discussion

Combining an operator with LRO$_2$ increases its effectiveness with a modest increase in the number of mutants generated. ORL and ORL$^+$ generate the biggest number of mutants and provide excellent coverage. ORL$^+$ always provides perfect or nearly perfect coverage. STL and

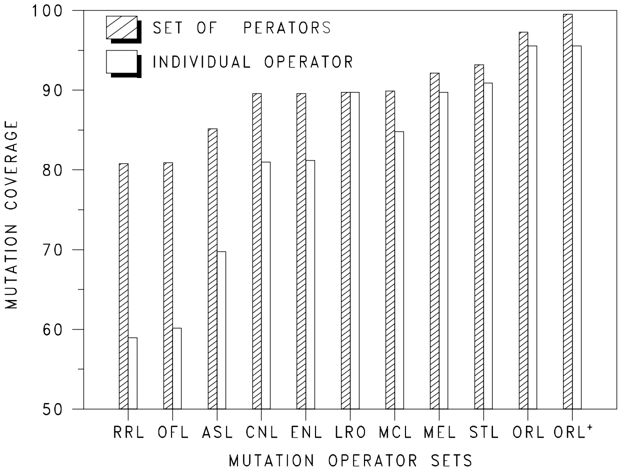


Figure 7.3: Average Mutation Scores of Operator Sets and Corresponding Individual Operators for the State-based Specifications.

ORO	STO	MEO	MCO	LRO	ENO	CNO	ASO	OFO	RRO
95.5	90.9	89.7	84.8	89.7	81.2	81.0	69.8	60.1	59.0

ORL	STL	MEL	MCL	LRO	ENL	CNL	ASL	OFL	RRL
97.3	93.2	92.1	89.9	89.7	89.6	89.6	85.1	80.9	80.8

Table 7.13: Average Mutation Coverage of Mutation Operators and Sets for the State-based Specifications.

MEL get second best coverage while generating far fewer mutants. Combining with LRO_2 results in a more pronounced improvement for the less effective operators. For example, while the gap in effectiveness between missing clause operator (MCO) and clause negation operator (CNO) is moderate, the gap between MCL and CNL is very small.

The results for the TCAS II/Boolean specifications differ slightly from the results for the state-based specifications, however, the relative merits of the mutation operators are similar.

7.5 Pairwise Coverage

In the previous sections, we used mutation coverage to compare mutation operators and operator sets. In this section, we compare them using an approximation of pairwise coverage to get an independent indication of their quality.

Pairwise testing was described in Section 2.1.3. We introduce an approximation of pairwise coverage for SMV specifications here. Let x and y be two variables in an SMV specification. Let the domain of x have values $c_1 \dots c_n$, the domain of y have values $d_1 \dots d_m$, where $n > 1, m > 1$.

ORO	STO	MEO	MCO	LRO	ENO	CNO	ASO
95.9	93.7	94.2	81.1	80.9	77.1	75.6	65.0

ORL	STL	MEL	MCL	LRO	ENL	CNL	ASL
99.7	97.6	97.6	87.4	80.9	84.0	84.0	81.8

Table 7.14: Average Mutation Coverage of Mutation Operators and Sets for TCAS II/Boolean.

Then the following set of requirements will cover all combinations of values for x and y:

$$AG(!(x = c_i)|!(y = d_j)),$$

where $1 \leq i \leq n$, $1 \leq j \leq m$.

Note that if a variable is of type Boolean, its domain has two values, 0 and 1. For integer variables, it is possible to get a very large number of requirements; partitioning the domain would be necessary.

Let N be the total number of combinations of valid values for all pairs of variables. Let k be the number of combinations covered by a test set. The pairwise coverage of the test set is $\frac{k}{N}$.

In the case of state-based specifications, the same variable at different time steps can be considered as different variables. This implies that to cover all variables, we need to give tests that assign all possible values to all variables at all steps. However, the number of time steps can be infinite and any limit on the number of steps is arbitrary. In designing the technique of reflection, Ammann et. al [3] introduced extra variables representing the values from a previous state. Therefore, the sample SMV specifications contain sets of state variables for two time steps. Accordingly, our approximation of pairwise coverage is a strong measure as it requires tests for all combinations of every pair of variables for two time steps.

In Table 7.15, we present pairwise coverage of all mutation operators, individual mutation operators, and mutation operator sets for cruise control example.

All	ORO	STO	MEO	MCO	LRO	ENO	CNO	ASO
98.2	95.6	93.8	95.1	93.0	95.4	77.6	75.8	88.7
	ORL	STL	MEL	MCL	LRO	ENL	CNL	ASL
	97.7	96.4	96.4	95.6	95.4	93.3	93.3	94.6

Table 7.15: Pairwise Coverage of Mutation Operators and Sets for Cruise Control.

We also evaluated pairwise coverage for the set of 20 specifications from TCAS II/Boolean. In Table 7.16, we present average pairwise coverage of all mutation operators, individual mutation operators, and mutation operator sets for TCAS II/Boolean specifications. The details for each individual specification can be found in Appendix B.

All	ORO	STO	MEO	MCO	LRO	ENO	CNO	ASO
94.4	91.8	81.2	81.9	79.5	82.8	70.7	69.0	65.2
	ORL	STL	MEL	MCL	LRO	ENL	CNL	ASL
	94.0	86.7	86.7	86.3	82.8	83.8	83.8	82.9

Table 7.16: Average Pairwise Coverage of Mutation Operators and Sets for TCAS II/Boolean.

Kobayashi et. al [42] performed the reverse experiment, where tests are generated using several test generation techniques, including pairwise testing, and then evaluated using mutation coverage. They generated tests for the 20 transition specifications in TCAS II/Boolean. They used the mutation operators ORO, ENO, LRO, and ASO for the mutation coverage metric. Their experiments show that pairwise testing gets low mutation coverage. We must note that the number of tests generated by pairwise testing (cost of testing) is also low.

Discussion

Specification-based mutation analysis gets high pairwise coverage. This implies that mutation generates tests which tend to cover the entire domain of the system. This gives us more confidence in specification-based mutation.

As Tables 7.15 and 7.16 show, ORL gets pairwise coverage which is almost as high as that of all operators combined. The relative pairwise coverage of different mutation operators follows the pattern that we noticed earlier for their relative mutation coverage.

In the next section, we point out a mutation operator that can achieve almost perfect pairwise coverage.

7.6 Getting Almost Perfect Pairwise Coverage

In Section 6.5, we theoretically compared clause insertion operator (CIO) and pairwise testing for Boolean specifications. In particular, if a test detects a CIO mutation, it will also cover a pair of value assignments for the corresponding pair of variables. However, there is no guarantee that such a test for a particular CIO mutation exists. We evaluate the pairwise coverage of CIO for the 20 Boolean specifications from TCAS II/Boolean.

Table B.8 in Appendix B presents the number of mutants, number of tests and pairwise coverage for clause insertion mutation operator for every specification as well as the averages. As noted above, this operator generates a very large number of mutants and a large number of tests. The perfect pairwise coverage is achieved for 9 out of 20 specifications, while the average coverage is 98.4%.

One of the specifications, for which the perfect pairwise coverage was not achieved, is specification number 20 (see Figure B.1):

$$S = \bar{e}f\bar{g}\bar{a}(bc \vee \bar{b}d)$$

The following pairs are not covered by the CIO adequate test set:

$$(a.1, e.1), (a.1, f.0), (a.1, g.1), (e.1, f.0), (e.1, g.1), (f.0, g.1)$$

Consider the pair $(a.1, e.1)$. There is one occurrence of a and one occurrence of e in the specification. This corresponds to case 2 of Theorem 4. The detection condition for the clause conjunction operator mutation where e is replaced with $e \wedge \bar{a}$ is

$$dS^{\mathrm{e}}_{\mathrm{e}\wedge\bar{\mathrm{a}}} = eaf\bar{g}\bar{a}(bc \vee \bar{b}d) = 0,$$

so there is no test for this mutation. Similarly, there is no test for the CCO mutation where a is replaced with $a \wedge \bar{e}$.

7.7 Summary

We evaluated the relative merits of mutation operators and combinations of mutation operators using mutation coverage and pairwise coverage. The effectiveness of operators depends partly on the form of the temporal logic formulas, in particular, having longer formulas with more logical connectors tends to increase the relative effectiveness of associative shift operator (ASO) and logical connector reference operator (LRO). The experimental results support our hypothesis that even though the hierarchy of mutation operators proved in Section 5 does not generally imply subsumption, harder-to-detect mutation operators generate tests which are more effective than those generated by easier-to-detect mutation operators.

We found that combinations of operators proposed in Section 3.3 are considerably more effective than the corresponding individual mutation operators with only a moderate increase in the number of mutants generated. ORL$^+$ gets the best coverage but generates a lot of mutants. MEL and STL have good coverage and generate far fewer mutants, so we suggest their use as an alternative to ORL$^+$ when the latter is prohibitively expensive.

The mutation operators get good pairwise coverage. Clause insertion operator (CIO) gets almost perfect pairwise coverage but generates a very large number of mutants. Now we turn our attention to the use of specification mutation for testing programs.

Chapter 8

Guaranteeing Fault Visibility

Specification-based testing is a black-box technique, that is, it assumes that the code is not known. Thus, failures in the code can only be detected in external responses. While a test produced using specification-based mutation testing will catch the corresponding failure in some state, there is no guarantee that the test will cause a visible failure. This reduces the usefulness of a mutation-adequate test suite produced using reflection, for example. We describe a method producing tests that guarantee fault visibility.

8.1 State Machine (SM) Duplication

Suppose a model checker compares the external behavior of the original and mutated state machines. Any counterexamples produced must exhibit failures, that is, inputs must be chosen to manifest differences in the outputs. To facilitate this comparison, we begin by duplicating the state machine and insuring that the duplicate always takes the same transition as the original. Then we can mutate the duplicate to implement the mutation testing criterion.

More formally, let SM be the description of the original state machine. Let SM_d be a duplicate of SM containing a mutation, or syntactic change. SM and SM_d have separate sets of output variables. We combine the two machines into a single state machine SM^+. We then assert that the values of the outputs of SM and SM_d are identical over SM^+. If SM_d has an observable fault, the model checker will produce a counterexample leading to the state where SM and SM_d differ in a value for the output.

From that counterexample, we can construct a test case containing values for inputs and the expected values for the outputs from the original state machine, SM.

8.2 Handling Nondeterminism

If the specification allows nondeterministic behavior, the expected outputs might not be adequate as an oracle. Nevertheless, the tests are expected to cause some faulty implementations to exhibit failures.

If there are any nondeterministic transitions in the original state machine, SM and a naively duplicated SM_d embedded in SM^+ are allowed to make different choices. For example, the statement in Figure 8.1 means that the next value for `var` may be either 1 or 2 if `condition` is true.

Nondeterminism is expressed in the state machine description language of SMV by giving a set of values for the result of an expression. When a variable is assigned a set of values, all possible values are explored independently of each other. If SM is duplicated naively, SMV could provide a counterexample that chooses one value of a variable in SM and another value of the corresponding variable in SM_d, that is, the "difference" arises from accidental differences

```
next(var) := case
  condition : {1, 2};
  1 : 0;
esac;
```

Figure 8.1: Nondeterminism in SMV

or differences in execution, not from semantic differences. We must force SM and SM_d to make the same choices when they have a nondeterministic choice. We achieve this by declaring a new variable globally for each nondeterministic choice. We modify both SM and SM_d to choose depending on this common global variable.

For the assignment statement in Figure 8.1, we declare a common unconstrained variable: `coin : {1, 2};`. We then modify both SM and SM_d to have this statement:

```
next(var) := case
  condition : coin;
  1 : 0;
esac;
```

While this method is general, it is excessive for variables without explicit transition, such as inputs: there are still no guards or formulas to mutate. In this case, we can simply move declarations of such variables into the `main` module and pass them to SM and SM_d as parameters.

8.3 An Illustrative Example

We use the example model in Figure 8.2(a), derived from [67], to illustrate the method of SM duplication. Variables `d`, `b`, and `f` are inputs and are not constrained. The variables `e` and `a` are intermediate variables. The statement `init(e) := 0;` sets `e` to 0 initially. The next value of `e` is `1` if the guard `f = On` is true, otherwise it is `0`. We consider the output to be the variable `out`, which has possible values `Low` and `High`. Its value is `High` if `a` is greater than 10, otherwise it is `Low`.

As Figure 8.2(b) illustrates, we rename `main` to `original`[1], move declarations of input variables into the new `main` module, instantiate the `original` and `duplicate` modules (SM and SM_d, respectively) in the new `main`, and pass inputs as parameters. If we wish to avoid passing each parameter separately, we can use a feature of SMV that allows to pass an instance of a module (`main` in this case) as a parameter.

The CTL formula asserts that outputs of the original and mutant modules are always the same. If there are several output variables, the assertions can be given in different ways, such as in Figure 8.3. If there is one `SPEC` formula for each output, as in Figure 8.3(b), more counterexamples are likely. The conjunction in Figure 8.3(a) makes the model checker find one counterexample for each mutant. That counterexample needs only have one output differ between the original and the mutant. In contrast, with one formula per output, the model checker tries to find a counterexample for each output for each mutant. Since a mutant rarely affects all outputs, counterexamples would not be found for all mutants and outputs. We have not investigated the number of unique counterexamples produced or differences in coverage from the two styles.

Assignment statements in the `duplicate` module from Figure 8.2(b) are candidates for mutation. Since we mutate the state machine description instead of temporal logic formulas, some mutations may result in a semantically invalid SMV model. Two cases are common. First, a mutation operator replacing one variable with another may generate a mutant containing a

[1] All modules of the original state machine description must be renamed for duplication.

```
MODULE main
VAR
  d: 0..5; b: 0..11; f: {On, Off};
  out: {Low, High};
  a: 0..16; e: 0..1;
ASSIGN
  init(e) := 0;
  next(e) := case
    f = On : 1;
    1 : 0;
  esac;
  a := e * d + b;
  out := case
    a > 10 : High;
    1 : Low;
  esac;
```

(a) An SMV Example.

```
MODULE original(d, b, f)
VAR
  out: {Low, High};
  a: 0..16; e: 0..1;
ASSIGN
... same transitions as in Figure 7 ...

MODULE duplicate(d, b, f)
VAR
  out: {Low, High};
  a: 0..16; e: 0..1;
ASSIGN
... same as original, to be mutated ...

MODULE main
VAR
  d: 0..5; b: 0..11; f: {On, Off};
  good : original(d, b, f);
  mutant : duplicate(d, b, f);

SPEC AG (good.out = mutant.out)
```

(b) After Duplication.

Figure 8.2: A Duplication Example.

```
SPEC AG (good.out1 = mutant.out1 & good.out2 = mutant.out2 & ...)
```

(a) A Combined Formula

```
SPEC AG (good.out1 = mutant.out1)
SPEC AG (good.out2 = mutant.out2)
              ...
```

(b) One Formula per Output

Figure 8.3: Specifying Multiple Outputs.

circular dependency. Our tools use SMV's built-in analysis to automatically remove such mutants from further consideration. Second, the value of an expression on the right hand side of an assignment in the mutant may be outside of the range of the variable on the left hand side. Consider a mutant of an assignment for variable a in Figure 8.2(a):

```
a := e * (d + 1) + b;
```

The right hand side of the mutant may evaluate to a value that is greater than the maximum allowed value of a, which was declared to be 16. To fix this, we change the declaration of a in the mutant to expand its range when needed.

8.4 Duplicating Processes

The example only shows synchronous composition of modules. In case of interleaving, introduced by the keyword process in SMV, special care must be taken to ensure that the processes of original and duplicate machines follow each other in an orderly fashion.

We can assign the original and mutant processes unique id numbers, for instance, 0 and 1. We pass Boolean variables, turn and valid, to the processes. Turn is initially 0. Each process changes it so that on the next step it is equal to the id of the other process. Variable valid becomes false if the processes are ever executed out of order, thus telling SMV to disregard other orderings.

The following CTL formula asserts that outputs of the original and mutant modules are the same after the second process executes, if the processes executed in order.

```
AG (turn = 0 & valid -> good.out = mutant.out)
```

8.5 Sharing Independent Variables

Some parts of the model may not depend on the variable affected by a particular mutation. These parts do not need to be duplicated. Strictly speaking, for any particular mutation, we need only duplicate the variable whose assignment is being mutated and any dependent variables. Dependency determinations can stop at output variables. Such dependency can be determined using program slicing [78]. If the model has many modules, only the module with the mutation and any dependent modules need to be duplicated. For large models with limited feedback, this may save enough model checking time to be worth the dependency analysis.

In Section 9.1, we evaluate the effectiveness of SM duplication at detecting seeded faults in an implementation of TCAS/Siemens.

Chapter 9

Program-Based Coverage

A comparison of the specification-based mutation analysis with commonly accepted criteria is needed. Up to this point, we did not consider the actual programs corresponding to the formal specifications. However, our goal is to reduce the number of faults in the programs. Therefore, we study usefulness of the tests generated from formal specifications for detecting bugs in the corresponding implementations. Additionally, we evaluate the program-based structural coverage of the tests.

We use two C programs for our experiments: TCAS/Siemens and Cruise Control. They were described in Section 7. Cruise control was implemented by Jeff Offutt. It has 6 procedures and 258 non-blank non-comment lines of code. TCAS/Siemens has 9 procedures and 135 non-blank non-comment lines of code.

We chose cruise control because it is a reactive system commonly studied in software testing. We chose TCAS/Siemens because it is a transformational system with large intermediate state. Both programs were written by others and come with sets of faulty versions. This makes our experiments more objective. Block and decision coverage of specification-based tests for a different implementation of cruise control (in Java) was examined in [3].

9.1 Effectiveness in Detecting Faults

Our goal is to reduce the number of faults in programs. Therefore, we ran experiments to evaluate the effectiveness of the methods for detecting seeded faults. TCAS/Siemens program comes with 39 faulty versions derived by manually seeding realistic faults. 26 versions have single mutations such as replacing a constant with another constant, replacing $\geq$ with $>$, or dropping a condition. The rest involve either multiple changes or more complex changes. Cruise control program comes with 25 faulty versions; the faults were inserted by Jeff Offutt [57].

We use ORL$^+$ to generate tests. ORL$^+$ is a combination of operand reference operator (ORO), LRO$_2$ (recall that LRO$_2$ replaces $\vee$ with $\oplus$, $\wedge$ with $\leftrightarrow$, $\rightarrow$ with $\leftrightarrow$), relational connector reference operator (RRO), and off-by-1 operator (OFO). For TCAS/Siemens, we compare two methods: guard reflection and SM duplication. For cruise control, we use guard reflection. We explain guard reflection in Section 2.6.1 and SM duplication in Section 8.

	Method	Mutants	UTs	Coverage
TCAS/Siemens	Guard Refl.	591	81	59%
	SM Dupl.	273	56	100%
Cruise control	Guard Refl.	508	45	76%

Table 9.1: Effectiveness in Detecting Seeded Faults.

In Table 9.1, "Mutants" is the total number of syntactically valid mutants, including consistent and duplicate mutants, "UTs" is the number of unique counterexamples or tests after duplicates and prefixes of longer counterexamples are removed. "Coverage" is the number of faulty versions detected by the method divided by the total number of faulty versions.

For TCAS/Siemens, we use NIST's Test Assistant for Objects (TAO) [8] to turn the counterexamples into concrete test cases. When provided with the correspondence between specification variables and function calls on the implementation level, TAO generates code to create new test instances, call the interface functions to set and get values, make sure the specified conditions hold, and report any differences between produced and expected results.

For cruise control, we wrote Perl scripts to turn the counterexamples into concrete test cases.

Table 9.1 shows that, for TCAS/Siemens, SM duplication detects 100% of faulty versions while guard reflection detects only 59% of the faults. We attribute the magnitude of the difference to a relatively large intermediate state of the program. The SM duplication method generated fewer mutants and test cases than guard reflection, yet it is much more effective in detecting seeded faults. The method duplicates the state machine thus increasing the size of the state space. The TCAS/Siemens specification is relatively small, so the limits of scalability have not been addressed.

The time required to generate tests for TCAS/Siemens using the guard reflection method was 3.5 seconds; SM duplication took 9 seconds. We used a 1.7 GHz Pentium 4[1] PC with 1 GB of RAM running Red Hat Linux. The SM duplication method took considerably longer due to the overhead of starting SMV and building the state machine model for every new mutant.

For cruise control, 5 of 25 seeded faults are not related to the functional specifications. The program has two modes set by a global option. In the first mode, the program always starts in the initial state, Off, and only the changes in test inputs can lead to other states (Inactive, Cruise, or Override). In the second mode, the test explicitly sets the state. We did not generate any tests that explicitly set the state, so those 5 seeded faults cannot be found. The two remaining faults are not found because of the difference in details of the semantics of the specification and the program. The program allows to set a variable without triggering an event. For instance, while in the state Cruise, the program can change the variable Ignited to False and still remain in Cruise. In contrast, in the SMV specification, changing Ignited to False will always trigger an event that will change the state to Off.

9.2 Structural Coverage Results

We use the tests generated using the set of mutation operators ORL$^+$, as in Section 9.1. For TCAS/Siemens, we use the method of SM duplication, whereas for cruise control, we use guard reflection.

We apply several structural measures: block, decision, C-use, and P-use coverage. These measures are defined in Section 2.1. The coverage is measured using ATAC [37]. We present the results in Table 9.2.

For TCAS/Siemens, 1 of 106 blocks is unreachable. 5 of 50 decisions are infeasible. For example, in a fragment

```
Own_Above_Threat() && Cur_Vertical_Sep >= MINSEP
  && Up_Separation >= ALIM()
```

the value of Cur_Vertical_Sep >= MINSEP is never false, since this fragment can be executed only if Cur_Vertical_Sep > MAXALTDIFF (MAXALTDIFF is greater than MINSEP).

The test cases cover 99 blocks and 44 decisions. The remaining 6 blocks and 1 decision are in the function *main* and they get executed only when there is a wrong number of input parameters. So these blocks and the decision are not related to the specification, that is, the specification does

[1] Pentium is a registered trademark of Intel Corporation.

		Block	Decision	C-use	P-use
TCAS/Siemens	total	106	50	43	34
	feasible	105	45	42	31
	covered	99	44	42	30
	coverage (%)	94.3	97.8	100	96.8
Cruise Control	total	186	173	45	150
	feasible	184	172	45	149
	covered	155	125	34	103
	coverage (%)	84.2	72.7	75.6	69.1

Table 9.2: Structural Coverage.

not model the case of wrong number of arguments. 42 of 43 C-uses are covered, 30 of 34 P-uses are covered. All of the uncovered C-uses and P-uses are in the uncovered blocks/decisions.

For cruise control, 2 of 186 blocks are unreachable. 155 blocks were covered. The remaining blocks were not related to the functional specification. In particular, they are concerned with a command option, checking for error opening a file, interactive mode, and checking for incorrect input variable name. The remaining decisions, C-uses, and P-uses were not covered because of the differences in details of the semantics of the program and the specification.

In summary, these experiments show that tests generated by specification-based mutation cover the parts of the implementation corresponding to the specification thoroughly. If the program has large intermediate state, the method of SM duplication is much more effective for detecting seeded faults. The program-based coverage of tests derived from specifications depends on the degree of the difference between the specification and the program.

Chapter 10

Conclusions

We believe that this work supports our hypothesis that specification-based mutation analysis can be used to economically generate effective tests. The main contributions of this thesis are as follows:

- We define an extensive set of mutation operators and implement a mutation generator tool for SMV specifications.
- We construct detection condition for a mutation, which replaces a subpredicate X with another predicate E in specification S, as a conjunction of origination condition and propagation condition:

  $$dS_{\mathrm{E}}^{\mathrm{X}} = (X \oplus E) \wedge \frac{dS}{dX}$$

  This allows us to prove that the hierarchy of mutation operators holds for arbitrary predicates, not just those in disjunctive normal form.
- We use our theoretical technique to extend the hierarchy of mutation operators. Additionally, our technique can be used in the future to compare other mutation operators.
- Based on analysis and empirical evaluation, we recommend mutation operators and sets of mutation operators that yield good test coverage at a reduced cost. In particular, ORL^{+} provides the best coverage while generating many mutants. STL and MEL get second best coverage while generating far fewer mutants.
- We evaluate program-based coverage of tests generated by specification-based mutation; the tests cover the parts of the program corresponding to the specification thoroughly.
- To improve the effectiveness for programs with large intermediate state, we devise a method which uses a model checker to guarantee that tests cause detectable output failures. The method is thorough at detecting typical program faults.
- We use our theoretical technique to analyze existing testing methods, in particular, the basic meaningful impact strategy [82] is stronger in that it tests for stuck-at mutations and not variable negation mutations as proposed by the authors.
- We find that, for Boolean specifications, clause insertion mutation operator gets almost perfect pairwise coverage.

Appendix A

Mutation Generator

To study the mutation operators, we developed an extensible tool for systematically making small syntactic changes to SMV [48] specifications. In what follows, "SMV" refers to SMV version 2.5 from Carnegie Mellon University, available at http://www.cs.cmu.edu/~modelcheck.

A.1 Overview

The mutation tool uses portions of SMV code: the parser, abstract syntax tree manipulation routines and low level functionalities, such as dynamic memory allocation and manipulation of data structures (e.g., hash tables).

Mutation generator performs the following steps:

1. Parse a given SMV file and build a tree data structure in memory.
2. Process the tree to extract information necessary for performing mutations, e.g., collect information about types and domains of variables.
3. For each selected mutation operator, traverse the tree invoking the corresponding mutation routine. When the routine recognizes an opportunity for a mutation, it creates a mutant. The mutant is then written to a file.

Resulting individual mutants may be left in individual SMV files, written to a single file, or divided between several files of size no larger than a user defined value. Leaving all mutants in individual files yields a large number of files. The overhead of starting a new SMV process for each mutant is intolerable even for specifications of moderate size. Since SMV builds a state machine transition relation for a given input file only once and checks CTL formulas independently, using an option that writes mutations into a single file results in very efficient processing.

The tool allows the user to selectively apply mutation operators. It can be extended with new operators. In addition, the mutation generator optionally mutates state machines to generate tests which a correct implementation should fail. The source code and documentation are available from the author.

A.2 Requirements

Since many mutants are usually generated and mutant generation is only a part of a larger testing process, efficiency is a very important general requirement.

The mutation engine had to satisfy the following requirements.

- Generate only syntactically and semantically legal mutants.

  It is theoretically possible to use SMV to check whether a mutant is legal. However, this is undesirable as SMV may take a long time to execute or may crash on an invalid statement. The mutation engine tries to predict and avoid generating illegal mutants, in some cases preferring to err on the side of not generating some legal mutants. For example, if there is a chance that a mutation will result in division by 0, we do not generate a mutant.

  Note that a state machine mutation may result in an invalid mutant, if restrictions on the structure of the "case" statements are not met.

- Allow users to apply the mutation operators selectively.

- For efficient processing, output all mutants to one file, but also retain an option to write each mutant in an individual file.

- Provide an option to generate mutant SPECs directly from the guards found in the ASSIGN declarations.

- Optionally generate failing tests by mutating state machines.

- For flexibility, provide many additional options, such as the ability to avoid mutating variables of a certain type.

- For efficiency, determine variable type without instantiating its container module.

  This restriction is necessary because a specification may contain many instances of a module. Since actual parameters for different instances of the same module may have different types, it becomes impossible to determine some variable types correctly. In these rare cases, some legal mutants may not be generated.

- Minimize the memory requirements.

  Since a parse tree may be very large, avoid storing a lot of information with every node of the parse tree.

- Optimize the speed of the mutant generation component.

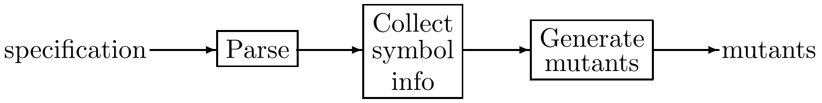
  Since the parse tree is traversed once for every mutation operator, the type of expressions corresponding to each node should not be determined on the fly, instead, it should be precomputed and stored in the parse tree. This is at odds with the previous requirement.

- In order to speed up the development, reuse existing code.

  We reused the parser and low level data structures with associated routines: storage management, tree nodes, hash tables.

A.3 Components

Interaction of the tool components is pictured below.



When given a specification, the tool first parses it to create an abstract syntax tree and save the blocks of code which are not mutated. Then the symbol information is collected from the parse tree. Additionally, the tool identifies the locations in the parse tree of the declarations to be mutated. After that, the mutants are generated by applying a set of mutation operators.

A.3.1 Data Structures

After parsing an input SMV specification, the tool builds a symbol table. Since SMV represents the state machine transition relation by ordered binary decision diagrams (OBDD) [13], it encodes every variable as a collection of Boolean variables. Mutation generator, however, uses type information to perform mutations. Hence, we designed the symbol table data structures and associated routines specifically for the mutation engine.

A module table contains an entry for each MODULE declaration. Each module table entry contains, in particular, a symbol classification table with an entry for each symbol in the module: a variable (from a VAR declaration), a macro (from a DEFINE declaration), a formal argument.

For effeciency, both module table and symbol classification table are implemented using SMV's hash table data structure.

A.3.2 Parsing

Parsing functionality, a lexical analyzer and a parser, was provided in the Carnegie Melon University's version of SMV.

We modified the SMV's lex source code to recognize the parts of input file which are not mutated. To read and write those parts in correct order, we save the starting line numbers of new declarations as they appear in the SMV file. We used the SMV context-free grammar (yacc productions) without change.

First, the parse tree is generated using the functions from SMV. After that, the program reads the blocks of input which are not mutated and saves them in an array.

The program expects a valid SMV specification as an input; therefore, it does not check for semantic errors. Most error messages are lexical or syntactical.

A.3.3 Mutant Generation

SPEC and ASSIGN are the declarations considered for mutation. Keyword ASSIGN introduces a collection of parallel assignments which determine the transition relation of the state machine and its initial state (or states). The specification of the system appears as a formula in CTL under the keyword SPEC.

By default, the SPECs found in an SMV file are mutated. If the SMV specification lacks SPECs, these can be generated by reflecting [2] the state machine declaration.

Optionally, the mutation engine can implement mutations of guards found in the ASSIGN declarations as SPECs, e.g., if ASSIGN declaration contains the following:

```
next(a) := case
  b1 : v1;
  b2 : v2;
    ...
esac;
```

then generate temporal formulas of the form: `SPEC AG (b1 ↔ b1')`, where b1' is a mutant of b1. b1, ... are called guards and v1, ... are called targets. b1, b2, ... must form a partition, v1, v2, ... must be pairwise disjoint, "case" statements must be flat.

Finally, the tool can apply mutation operators directly to the state machine (guards and targets of the ASSIGN declarations).

Basic tree operations

To create a mutant, a subtree of the parse tree is modified by applying one of several subtree operations. This is followed by outputting the mutant. After that, an inverse operation is applied to the mutant to restore the original tree. The process is repeated for each mutant.

Below is a list of the basic tree operations. After each operation, we give examples of mutation operators relying on the operation. Here, we classify mutation operators based on the underlying tree operations.

1. Replace a leaf node with another leaf node. The operation is an inverse of itself.

   Operand reference operator (ORO) replaces a symbol name contained in a leaf node with another name. The symbol name represents an operand.

   Examples of ORO are variable replacement, constant replacement, constant set element replacement, user specified token replacement.

2. Replace a subtree (or a leaf node) with a leaf node.

   Stuck-at operator replaces a subtree representing a clause with a leaf node representing 0 (false) or 1(true).

   For the array index reference operator, a subtree representing an array index is replaced with a node representing an array bound or an incremented or decremented value of the index.

3. Change the type of an internal node. The operation is an inverse of itself.

   Relational, arithmetic, and logical connector reference operators replace a connector with another connector.

4. Enlarge a subtree, that is, replace a node with a larger subtree containing the original node as its subtree.

   Off-by-1 operator replaces a node representing an arithmetic expression e with a node representing $e + 1$ and a node representing $e - 1$.

   Clause negation operator and expression negation operator insert a “negation” node between a node representing a predicate and its parent node.

   “Contract”, below, is the inverse operation.

5. Contract a subtree, that is, replace an internal node with its child.

   Missing clause operator and missing expression operator replace a subtree representing an expression with one of its children. The other child is the removed expression.

   Other operators using this basic tree operation are removal of a subexpression from an arithmetic expression, and integer set element removal.

   “Contract” operation is an inverse of “enlarge” operation.

6. Rotate a subtree. Left and right rotations are inverse operations.

   Associative shift mutation rotates a subtree.

7. Swap subtrees. The operation is an inverse of itself.

   Operand interchange mutation swaps operands of a non-commutative operation.

Appendix B

Details of Experiments for TCAS II/Boolean

Figure B.1 lists the set of transition specifications reported in [82]. We call this set of specifications TCAS II/Boolean. It originates from the specification of TCAS II, an aircraft collision avoidance system described by Leveson et. al [46]. The variables represent clauses, such as Own-Tracked-Altitude $\geq$ 15500.

Table B.1 presents the number of variables, as well as the number of mutants and the number of unique traces generated by applying all mutation operators to TCAS II/Boolean specifications.

We evaluated mutation coverage for TCAS II/Boolean specifications. Table B.2 gives the number of mutants for individual mutation operators, as well as the distribution of mutants by operator. Table B.3 gives the number of tests generated by individual mutation operators. Table B.4 presents mutation coverage of individual mutation operators. Table B.5 presents mutation coverage of each mutation operator combined with LRO_2 operator.

We also evaluated pairwise coverage for TCAS II/Boolean specifications. Table B.6 presents pairwise coverage of all mutation operators combined and of individual mutation operators for every specification as well as the average coverage. Table B.7 presents pairwise coverage of each mutation operator combined with LRO_2 operator. Finally, Table B.8 presents the number of mutants, number of tests, and pairwise coverage for clause insertion mutation operator for every specification as well as the averages.

1 $\overline{(ab)}(d\bar e\bar f \vee \bar d e\bar f \vee \bar d\bar e\bar f)(ac(d \vee e)h \vee a(d \vee e)\bar h \vee b(e \vee f))$

2 $(a((c \vee d \vee e)g \vee af \vee c(f \vee g \vee h \vee i)) \vee (a \vee b)(c \vee d \vee e)i)$
$\wedge\overline{(ab)}\ \overline{(cd)}\ \overline{(ce)}\ \overline{(de)}\ \overline{(fg)}\ \overline{(fh)}\ \overline{(fi)}\ \overline{(gh)}\ \overline{(hi)}$

3 $(a(\bar d \vee \bar e \vee de\overline{(\bar f gh\bar i \vee \bar g hi)}\ \overline{(\bar f glk \vee \bar g\bar i k)})$
$\vee\overline{(\bar f gh\bar i \vee \bar g hi)}\ \overline{(\bar f glk \vee \bar g\bar i k)}(b \vee c\bar m \vee f))(a\bar b\bar c \vee \bar a b\bar c\bar a\bar b c)$

4 $a(\bar b \vee \bar c)d \vee e$

5 $a(\bar b \vee \bar c \vee bc\overline{(\bar f gh\bar i \vee \bar g hi)}\ \overline{(\bar f glk \vee \bar g\bar i k)}) \vee f$

6 $(\bar a b \vee a\bar b)\overline{(cd)}(f\bar g\bar h \vee \bar f g\bar h \vee \bar f\bar g\bar h)\overline{(jk)}((ac \vee bd)e(f \vee (i(gj \vee hk))))$

7 $(\bar a b \vee a\bar b)\overline{(cd)}\ \overline{(gh)}\ \overline{(jk)}((ac \vee bd)e(\bar i \vee \bar g\bar k \vee \bar j(\bar h \vee \bar k)))$

8 $(\bar a b \vee a\bar b)\overline{(cd)}\ \overline{(gh)}((ac \vee bd)e(fg \vee \bar f h))$

9 $\overline{(cd)}(\bar e f\bar g\bar a(bc \vee \bar b d))$

10 $a\bar b\bar c d\bar e f(g \vee \bar g(h \vee i))\overline{(jk \vee \bar j l \vee m)}$

11 $a\bar b\bar c(\overline{(f(g \vee \bar g(h \vee i)))} \vee f(g \vee \bar g(h \vee i))\bar d\bar e)\overline{(jk \vee \bar j l\bar m)}$

12 $a\bar b\bar c(f(g \vee \bar g(h \vee i))(\bar e\bar n \vee d) \vee \bar n)(jk \vee \bar j l\bar m)$

13 $a \vee b \vee c \vee \bar c\bar d ef\bar g\bar h \vee i(j \vee k)\bar l$

14 $ac(d \vee e)h \vee a(d \vee e)\bar h \vee b(e \vee f)$

15 $a((c \vee d \vee e)g \vee af \vee c(f \vee g \vee h \vee i)) \vee (a \vee b)(c \vee d \vee e)i$

16 $a(\bar d \vee \bar e \vee de\overline{(\bar f gh\bar i \vee \bar g hi)}\ \overline{(\bar f glk \vee \bar g\bar i k)})$
$\vee\overline{(\bar f gh\bar i \vee \bar g hi)}\ \overline{(\bar f glk \vee \bar g\bar i k)}(b \vee c\bar m \vee f)$

17 $(ac \vee bd)e(f \vee (i(gj \vee hk)))$

18 $(ac \vee bd)e(\bar i \vee \bar g\bar k \vee \bar j(\bar h \vee \bar k))$

19 $(ac \vee bd)e(fg \vee \bar f h)$

20 $\bar e f\bar g\bar a(bc \vee \bar b d)$

Figure B.1: Transition Specifications of TCAS II/Boolean.

	Var-s	Mutants	Traces
1	7	372	34
2	9	652	63
3	12	977	85
4	5	65	12
5	9	360	53
6	11	568	52
7	10	402	62
8	8	291	29
9	7	156	16
10	13	328	44
11	13	443	65
12	14	394	64
13	12	269	39
14	7	191	35
15	9	327	48
16	12	784	117
17	11	221	49
18	10	209	53
19	8	152	31
20	7	124	15

Table B.1: Number of Variables, Mutants, and Traces for TCAS II/Boolean.

Spec	ALL	ORO	STO	MEO	MCO	ENO	CNO	LRO	ASO
1	372	138	46	44	23	45	23	66	11
2	652	288	72	70	36	71	36	105	11
3	977	506	92	90	46	91	46	135	18
4	65	20	10	8	5	9	5	12	2
5	360	160	40	38	20	39	20	57	7
6	568	280	56	54	28	55	28	81	15
7	402	189	42	40	21	41	21	60	10
8	291	119	34	32	17	33	17	48	9
9	156	60	20	18	10	19	10	27	3
10	328	180	30	28	15	29	15	42	5
11	443	240	40	38	20	39	20	57	10
12	394	221	34	32	17	33	17	48	10
13	269	143	26	24	13	25	13	36	3
14	191	72	24	22	12	23	12	33	6
15	327	144	36	34	18	35	18	51	10
16	784	407	74	72	37	73	37	108	14
17	221	110	22	20	11	21	11	30	8
18	209	99	22	20	11	21	11	30	7
19	152	63	18	16	9	17	9	24	6
20	124	48	16	14	8	15	8	21	3
Avg	364.2	174.3	37.7	35.7	18.9	36.7	18.9	53.5	8.4
Prct	100.0	47.9	10.4	9.8	5.2	10.1	5.2	14.7	2.3

Table B.2: Number of Mutants for TCAS II/Boolean.

Spec	ALL	ORO	STO	MEO	MCO	ENO	CNO	LRO	ASO
1	34	30	16	19	12	16	13	17	7
2	63	49	27	29	14	20	18	28	8
3	85	78	21	22	14	20	17	22	8
4	12	9	8	7	5	6	5	7	2
5	53	45	18	18	13	16	14	17	6
6	52	45	21	24	18	20	15	21	6
7	62	55	20	21	16	17	15	23	7
8	29	23	15	17	13	14	11	15	6
9	16	11	11	12	10	7	6	10	3
10	44	35	16	19	13	11	8	16	4
11	65	55	19	19	16	11	11	20	7
12	64	56	17	18	14	12	9	19	7
13	39	31	15	15	12	7	7	18	3
14	35	28	13	16	10	12	9	16	6
15	48	38	17	18	12	12	10	22	7
16	117	101	38	39	28	32	28	34	11
17	49	42	15	18	11	15	11	13	7
18	53	43	14	17	10	13	9	15	7
19	31	26	12	14	9	12	9	10	5
20	15	11	10	10	8	6	5	9	3
Avg	48.3	40.5	17.1	18.6	12.9	13.9	11.5	17.6	6.0

Table B.3: Number of Tests for TCAS II/Boolean.

Spec	ORO	STO	MEO	MCO	ENO	CNO	LRO	ASO
1	98.7	94.3	95.5	87.9	91.1	89.8	87.9	76.4
2	96.9	94.1	94.3	81.6	78.8	78.5	87.3	65.5
3	96.6	90.8	91.6	67.7	85.7	82.8	83.2	55.9
4	92.9	95.2	92.9	88.1	83.3	81.0	90.5	54.8
5	96.3	93.2	93.2	74.7	88.3	86.4	85.2	55.6
6	97.1	92.5	93.4	89.2	87.1	82.2	83.0	65.6
7	96.8	94.1	94.5	82.7	82.7	81.8	88.6	68.2
8	95.6	93.4	94.8	89.0	85.3	83.1	82.3	73.5
9	93.3	93.3	94.7	89.3	81.3	80.0	82.7	72.0
10	94.7	94.7	96.5	88.8	72.9	71.2	78.8	51.8
11	96.4	91.6	91.6	83.1	64.3	64.3	72.3	58.6
12	97.0	92.8	93.2	82.7	69.6	68.3	74.7	74.7
13	95.7	90.8	90.8	84.7	59.5	59.5	81.0	46.0
14	96.0	95.2	96.8	91.9	79.8	78.2	85.5	67.7
15	95.5	91.0	91.6	78.7	61.2	60.7	89.3	70.2
16	97.1	94.8	95.0	70.4	82.0	80.5	72.0	50.1
17	96.5	96.5	97.1	71.8	61.2	60.0	66.5	78.2
18	94.3	94.3	95.5	75.8	75.2	73.2	73.9	67.5
19	96.4	96.4	97.3	62.7	79.1	77.3	73.6	73.6
20	94.2	94.2	94.2	81.2	73.9	72.5	79.7	73.9
Avg	95.9	93.7	94.2	81.1	77.1	75.6	80.9	65.0

Table B.4: Mutation Coverage of Mutation Operators for TCAS II/Boolean.

Spec	ORL	STL	MEL	MCL	ENL	CNL	LRO	ASL
1	100.0	96.8	96.8	92.4	93.6	93.6	87.9	85.3
2	100.0	97.5	97.5	86.4	87.3	87.3	87.3	79.9
3	100.0	94.5	94.5	81.5	90.8	90.8	83.2	80.7
4	100.0	100.0	100.0	97.6	92.9	92.9	90.5	81.0
5	99.4	96.3	96.3	81.5	92.0	92.0	85.2	82.7
6	99.6	95.0	95.0	92.1	89.6	89.6	83.0	83.8
7	99.5	97.7	97.7	86.8	86.4	86.4	88.6	81.4
8	99.3	97.1	97.1	92.7	87.5	87.5	82.3	89.0
9	100.0	100.0	100.0	97.3	88.0	88.0	82.7	90.7
10	100.0	100.0	100.0	94.1	79.4	79.4	78.8	82.3
11	99.2	94.8	94.8	86.3	69.5	69.5	72.3	74.3
12	100.0	96.6	96.6	87.3	74.7	74.7	74.7	86.9
13	99.4	96.3	96.3	90.2	75.5	75.5	81.0	67.5
14	100.0	100.0	100.0	97.6	84.7	84.7	85.5	76.6
15	100.0	95.5	95.5	83.7	77.0	77.0	89.3	76.4
16	99.6	97.3	97.3	76.6	84.9	84.9	72.0	69.6
17	99.4	99.4	99.4	84.1	81.8	81.8	66.5	87.1
18	98.7	98.7	98.7	80.9	83.4	83.4	73.9	80.9
19	99.1	99.1	99.1	70.9	81.8	81.8	73.6	88.2
20	100.0	100.0	100.0	87.0	79.7	79.7	79.7	91.3
Avg	99.7	97.6	97.6	87.4	84.0	84.0	80.9	81.8

Table B.5: Mutation Coverage of Mutation Operator Sets for TCAS II/Boolean.

Spec	ALL	ORO	STO	MEO	MCO	ENO	CNO	LRO	ASO
1	97.6	97.6	91.7	91.7	90.5	86.9	86.9	89.3	78.6
2	100.0	100.0	97.2	97.2	86.1	90.3	90.3	95.1	73.6
3	93.2	91.7	76.1	76.9	73.9	72.0	67.4	77.3	59.9
4	90.0	85.0	80.0	80.0	77.5	67.5	67.5	82.5	42.5
5	95.1	94.4	81.2	81.2	81.2	75.0	75.0	84.0	71.5
6	99.5	97.3	82.7	85.9	81.8	84.1	78.2	87.7	64.5
7	100.0	97.2	82.2	82.8	80.6	80.0	78.3	87.8	74.4
8	100.0	95.5	82.1	84.8	80.4	83.0	77.7	83.9	75.0
9	84.5	78.6	78.6	79.8	77.4	56.0	54.8	78.6	50.0
10	86.9	82.7	75.6	77.6	75.0	53.9	51.9	73.7	54.2
11	88.1	86.9	76.9	76.9	76.6	59.9	59.9	77.6	59.6
12	89.3	88.5	72.8	72.8	72.5	54.7	54.4	73.3	62.6
13	84.5	82.2	75.8	75.8	75.8	42.4	42.4	78.4	48.5
14	100.0	97.6	85.7	85.7	85.7	73.8	73.8	88.1	70.2
15	97.9	96.5	84.7	84.7	79.2	68.8	68.8	88.9	70.8
16	99.2	98.1	87.9	87.9	87.9	79.2	79.2	89.0	76.5
17	99.1	95.0	77.7	78.6	76.8	77.3	72.3	80.9	75.0
18	100.0	96.7	79.4	80.6	77.8	73.3	72.2	83.9	75.0
19	100.0	95.5	78.6	80.4	76.8	80.4	75.9	79.5	72.3
20	83.3	78.6	77.4	77.4	76.2	54.8	53.6	76.2	50.0
Avg	94.4	91.8	81.2	81.9	79.5	70.7	69.0	82.8	65.2

Table B.6: Pairwise Coverage of Mutation Operators for TCAS II/Boolean.

Spec	ORL	STL	MEL	MCL	ENL	CNL	LRO	ASO
1	97.6	94.0	94.0	94.0	91.7	91.7	89.3	88.1
2	100.0	97.9	97.9	95.1	95.8	95.8	95.1	93.1
3	93.2	78.4	78.4	76.5	78.0	78.0	77.3	76.5
4	90.0	90.0	90.0	90.0	85.0	85.0	82.5	77.5
5	95.1	85.4	85.4	85.4	84.7	84.7	84.0	82.6
6	98.6	90.5	90.5	90.5	87.7	87.7	87.7	87.3
7	98.9	89.4	89.4	89.4	88.9	88.9	87.8	88.9
8	98.2	88.4	88.4	88.4	85.7	85.7	83.9	87.5
9	84.5	84.5	84.5	84.5	78.6	78.6	78.6	79.8
10	86.9	80.8	80.8	80.5	74.7	74.7	73.7	77.6
11	88.1	79.8	79.8	79.5	77.2	77.2	77.6	78.2
12	89.3	78.6	78.6	78.3	73.3	73.3	73.3	77.5
13	84.1	81.8	81.8	81.8	78.0	78.0	78.4	78.8
14	100.0	89.3	89.3	89.3	88.1	88.1	88.1	85.7
15	97.9	89.6	89.6	88.2	84.0	84.0	88.9	82.6
16	98.9	90.9	90.9	90.9	90.2	90.2	89.0	89.4
17	98.2	87.7	87.7	87.7	86.4	86.4	80.9	84.1
18	98.9	88.9	88.9	88.9	87.8	87.8	83.9	86.1
19	98.2	86.6	86.6	85.7	84.8	84.8	79.5	81.2
20	83.3	82.1	82.1	81.0	76.2	76.2	76.2	76.2
Avg	94.0	86.7	86.7	86.3	83.8	83.8	82.8	82.9

Table B.7: Pairwise Coverage of Mutation Operator Sets for TCAS II/Boolean.

Spec	Mutants	UTs	Coverage
1	552	44	100.0
2	1152	82	100.0
3	2024	103	98.5
4	80	18	100.0
5	640	63	100.0
6	1120	66	99.1
7	756	83	99.4
8	476	32	99.1
9	240	16	92.9
10	720	68	93.0
11	960	136	98.4
12	884	139	98.9
13	572	93	97.3
14	288	49	100.0
15	576	70	99.3
16	1628	235	100.0
17	440	77	100.0
18	396	80	100.0
19	252	45	100.0
20	192	20	92.9
Avg	697.4	76.0	98.4

Table B.8: Pairwise Coverage of Clause Insertion Operator for TCAS II/Boolean.

Bibliography

[1] Paul Ammann, Paul E. Black, and Wei Ding. Model checkers in software testing. Technical Report NIST-IR-6777, U.S. National Institute of Standards and Technology, February 2002.

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

[3] Paul E. Ammann, Paul E. Black, and William 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, December 1998.

[4] Joanne 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, January 1996.

[5] Joanne M. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.

[6] Chonlawit Banphawatthanarak, Bruce H. Krogh, and Ken Butts. Symbolic verification of executable control specifications. In *Proceedings of the* 10^{th} *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.

[7] Ramesh Bharadwaj and Constance 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.

[8] Paul E. Black. Modeling and marshaling: Making tests from model checker counterexamples. In *Proceedings of the 19th Digital Avionics Systems Conference (DASC)*, volume 1.B.3, pages 1–6, Philadelphia, Pennsylvania, Oct 2000. IEEE.

[9] Paul E. Black, Vadim Okun, and Yaacov Yesha. Mutation of model checker specifications for test generation and evaluation. In *Mutation 2000 Symposium*, pages 24–30, San Jose, CA, October 2000.

[10] Paul E. Black, Vadim Okun, and Yaacov Yesha. Mutation operators for specifications. In 15^{th} *IEEE International Conference on Automated Software Engineering (ASE2000)*, pages 81–88, Grenoble, France, September 2000. IEEE Computer Society.

[11] G. Bochmann and A. Petrenko. Protocol testing: Review of methods and relevance for software testing. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis*, pages 109–124, 1994.

[12] R. Browmlie, J. Prowse, and M. S. Phadke. Robust testing of AT&T PMX/StarMAIL using OATS. *AT&T Technical Journal*, 71(3):41–47, May/June 1992.

[13] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, June 1992.

[14] John Callahan, Francis Schneider, and Steve Easterbrook. Automated software testing using model-checking. In *Proceedings of the 1996 SPIN Workshop*, Rutgers, NJ, Aug 1996. Also WVU Technical Report #NASA-IVV-96-022.

[15] William Chan, Richard J. Anderson, Paul Beame, Steve Burns, Francesmary Modugno, David Notkin, and Jon D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498 – 520, July 1998.

[16] T. Y. Chen and M. F. Lau. Test case selection strategies based on boolean specifications. *Software Testing, Verification and Reliability*, 11(3):165–180, September 2001.

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

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

[19] Edmund M. Clarke, Jr., E. Allen Emerson, and A. Prasad 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.

[20] Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, 15(11):1318–1332, November 1989.

[21] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–443, July 1997.

[22] Steve Cornett. *Code Coverage Analysis*. Bullseye Testing Technology, http://www.bullseye.com/coverage.html (accessed 15 January 2004).

[23] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[24] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[25] E. Allen Emerson and Joseph Y. Halpern. “Sometimes” and “not never” revisited: On branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, January 1986.

[26] André Engels, Loe Feijs, and Sjouke Mauw. Test generation for intelligent networks using model checking. In Ed 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.

[27] J. J. Filliben and A. Heckert. *Dataplot*. U.S. National Institute of Standards and Technology, http://www.itl.nist.gov/div898/software/dataplot (accessed 20 January 2004).

[28] Kenneth A. Foster. Error sensitive test cases analysis (ESTCA). *IEEE Transactions on Software Engineering*, 6(3):258–264, May 1980.

[29] Antony P. Galton, editor. *Temporal Logics and Their Applications.* Academic Press, 1987.

[30] Angelo Gargantini and Constance 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.

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

[32] Ajei Gopal and Tim Budd. Program testing by specification mutation. Technical Report TR 83-17, University of Arizona, November 1983.

[33] Tarak Goradia. *Dynamic Impact Analysis: Analyzing Error Propagation in Program Executions.* PhD thesis, Dept. of Computer Science, New York University, 1988.

[34] Tarak Goradia. Dynamic impact analysis: A cost-effective technique to enforce error-propagation. In *Proceedings of the 1993 International Symposium on Software Testing and Analysis*, pages 171–181, 1993.

[35] Kelly J. Hayhurst, Dan S. Veerhusen, John J. Chilenski, and Leanna K. Rierson. A practical tutorial on modified condition/decision coverage. Technical Report NASA/TM-2001-210876, NASA, May 2001.

[36] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Sung Deok Cha. Automatic test generation from statecharts using model checking. Technical Report MS-CIS-01-07, University of Pennsylvania, 2001.

[37] J. R. Horgan and S. A. London. ATAC: A data flow coverage testing tool for c. In *Symposium on Assessment of Quality Software Development Tools*, pages 2–10, New Orleans, LA, May 1992.

[38] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, July 1982.

[39] William E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, 2(3):208–215, 1976.

[40] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 191–200, May 1994.

[41] J. Kirby Jr. Example NRL/SCR software requirements for an automobile cruise control and monitoring system. Technical Report TR-87-07, Wang Institute of Graduate Studies, July 1987.

[42] Noritaka Kobayashi, Tatsuhiro Tsuchiya, and Tohru Kikuno. Non-specification-based approaches to logic testing for software. *Information and Software Technology*, 44(2):113–121, February 2002.

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

[44] D. Richard Kuhn. Fault classes and error detection in specification based testing. *ACM Transactions on Software Engineering Methodology*, 8(4):411–424, October 1999.

[45] M. F. Lau and Y. T. Yu. On the relationships of faults for boolean specification based testing. In *2001 Australian Software Engineering Conference*, pages 21–28. IEEE CS Press, August 2001.

[46] N. G. Leveson, M. P.E. Heimdahl, H. Hildreth, and J. Reese. Requirements specification for process control systems. *IEEE Transactions on Software Engineering*, SE-20(9):684–707, September 1994.

[47] Aditya Mathur. Performance, effectiveness, and reliability issues in software testing. In *Fifteenth Annual International Computer Software and Applications Conference*, pages 604–605, Tokyo, Japan, September 1991.

[48] Ken L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[49] Harlan D. Mills. On the statistical validation of computer programs. In *Software Productivity*, pages 71–81, Boston, 1983. Little, Brown. Also Technical Report FSC 72-6015, IBM Federal Systems Division, 1972.

[50] Larry J. Morell. *A Theory of Error-Based Testing.* Dissertation, Dept. of Computer Science, University of Maryland, August 1984.

[51] Larry J. Morell. Theoretical insights into fault-based testing. Contractor Report NASA CR-183277, NASA, 1988.

[52] S. Ntafos. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, 14(6):868–874, June 1988.

[53] A. J. Offutt, J. Voas, and J. Payne. Mutation operators for Ada. Technical Report ISSE-TR-96-09, George Mason University, October 1996.

[54] A. J. Offutt and J. M. Voas. Subsumption of condition coverage techniques by mutation testing. Technical Report ISSE-TR-96-01, George Mason University, January 1996.

[55] A. Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1(1):3–18, January 1992.

[56] A. Jefferson Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf. An experimental determination of sufficient mutation operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118, April 1996.

[57] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *Software Testing, Verification and Reliability*, 13(1):25–53, March 2003.

[58] Jeff Offutt, Yiwei Xiong, and Shaoying Liu. Criteria for generating specification-based tests. In *Proceedings of the Fifth IEEE Fifth International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, pages 119–131, Las Vegas, NV, October 1999. IEEE Computer Society Press.

[59] Vadim Okun, Paul E. Black, and Yaacov Yesha. Testing with model checker: Insuring fault visibility. In Nikos E. Mastorakis and Petr Ekel, editors, *Proceedings of 2002 WSEAS International Conference on System Science, Applied Mathematics and Computer Science, and Power Engineering Systems*, pages 1351–1356, Rio de Janeiro, Brazil, Oct 2002.

[60] Vadim Okun, Paul E. Black, and Yaacov Yesha. Testing with model checker: Insuring fault visibility. Technical Report NIST-IR, U.S. National Institute of Standards and Technology, July 2002.

[61] Vadim Okun, Paul E. Black, and Yaacov Yesha. Comparison of fault classes in specification-based testing. *Information and Software Technology*, 2004. To Appear.

[62] A. Petrenko, N. Yevtushenko, G. Bochmann, and R. Dssouli. Testing in context: framework and test derivation. *Special Issue on Protocol Engineering of Computer Communication*, 1997.

[63] M. S. Phadke. *Quality Engineering Using Robust Design*. Prentice-Hall, Englewood Cliffs, NJ, November 1989.

[64] A. Pnueli. The temporal logic of programs. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 46–57, New York, 1977.

[65] R. M. Poston. *Automating Specification-Based Software Testing*. IEEE Computer Society Press, 1996.

[66] A. Prior. *Past, Present, and Future*. Oxford University Press, London, 1967.

[67] Scott Ranville, 2002. Personal Communication.

[68] S. Rapps and E. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4), April 1985.

[69] Sanjay Rayadurgam and Mats P.E. Heimdahl. Coverage based test-case generation using model checkers. In 8^{th} *Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*, Washington, DC, April 2001.

[70] I. S. Reed. Boolean difference calculus and fault finding. *SIAM Journal Applied Mathematics*, 24(1):134–143, January 1973.

[71] D. J. Richardson and M. C. Thompson. An analysis of test data selection criteria using the relay model of fault detection. *IEEE Transactions on Software Engineering*, 19(6):533–553, June 1993.

[72] Gregg Rothermel and Mary Jean Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering*, 24(6):401–419, 1998.

[73] W. J. Spillman and E. Lang. *The Law of Diminishing Returns*. 1924.

[74] Kuo-Chung Tai. Theory of fault-based predicate testing for computer programs. *IEEE Transactions on Software Engineering*, 22(8):552–562, August 1996.

[75] Kuo-Chung Tai and Yu Lei. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering*, 28(1):1–3, January 2002.

[76] Kuo-Chung Tai, M. A. Vouk, Amit Paradkar, and P. Lu. Evaluation of a predicate-based software testing strategy. *IBM Systems Journal*, 33(3):445–457, 1994.

[77] Meyer C. Tanuan. Automated analysis of unified modeling language (UML) specifications. M. S. paper, University of Waterloo, 2001.

[78] F. Tip. A survey of program slicing techniques. *Programming languages*, 3:121–189, 1995.

[79] Tatsuhiro Tsuchiya and Tohru Kikuno. On fault classes and error detection in specification based testing. *ACM Transactions on Software Engineering Methodology*, 11(1):58–62, January 2002.

[80] K. S. How Tai Wah. A theoretical study of fault coupling. *Software Testing, Verification and Reliability*, 10(1):3–45, March 2000.

[81] Chang-Jia Wang and Ming T. Liu. A test suit generation method for extended finite state machines using axiomatic semantics approach. In R. J. Linn, Jr. and M.U. Uyar, editors, *Protocol Specification Testing and Verification, XII*, pages 29–43. Elsevier Science Publishers B.V. (North-Holland), 1992.

[82] Elaine Weyuker, Tarak Goradia, and Ashutosh Singh. Automatically generating test data from a boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, May 1994.

[83] Elaine J. Weyuker, Stewart N. Weiss, and Dick Hamlet. Comparison of program testing strategies. In *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification*, pages 1–10. ACM Press, 1991.

[84] W. E. Wong. *On Mutation and Data Flow.* PhD thesis, Purdue University, West Lafayette, IN, 1993.

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

[86] Steven J. Zeil. Perturbation techniques for detecting domain errors. *IEEE Transactions on Software Engineering*, 15(6):737–746, June 1989.

[87] Dan Zhou and Paul E. Black. Translating HOL to specifications for the model checker SMV. In *TPHOLs 2001, supplemental proceedings*, pages 400–415, Edinburgh, Scotland, September 2001.

[88] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.