Evaluation of Three Specification-based Testing Criteria

Aynur Abdurazik,[*] Paul Ammann,[†] Wei Ding,[†] and Jeff Offutt[*]
Department of Information and Software Engineering
Software Engineering Research Laboratory
George Mason University
Fairfax, VA 22030-4444 USA
{aynur, pammann, wding, ofut}@ise.gmu.edu

Abstract

This paper compares three specification-based testing criteria using Mathur and Wong's PROBSUBSUMES *measure. The three criteria are specification-mutation coverage, full predicate coverage, and transition-pair coverage. A novel aspect of the work is that each criterion is encoded in a model checker, and the model checker is used first to generate test sets for each criterion and then to evaluate test sets against alternate criteria. Significantly, the use of the model checker for generation of test sets eliminates human bias from this phase of the experiment. The strengths and weaknesses of the criteria are discussed.*

1. Introduction

There are many approaches to generating tests. There are also criteria to measure the completeness, adequacy, or coverage of tests on source code [27]. However, most test coverage criteria are either not objective or depend on a particular implementation. This paper uses Mathur and Wong's PROBSUBSUMES measure [18] to empirically compare three test coverage criteria on an example specification. The investigation is carried out with a model checker, the advantage being that human bias is eliminated from the resulting test cases.

There is an increasing need for high quality software, particularly for safety-critical applications such as avionics, medical, and other control systems. Developers have responded to this need in many ways, including improving the process, increasing the attention on early development activities, and using formal methods to describe requirements, specifications, and designs. Although all of these improvements help create software that is of higher quality and higher reliability, the software still needs to be tested, and the more stringent needs for the product also means that the testing method must be more effective at finding problems in the software. Project and test managers are more than ever in a position where they need solid information for how to apply scarce resources. Applying structured, precisely defined testing techniques allows development resources to be used more wisely.

Specification-based testing refers to creating test inputs from the software specifications. Specification-based testing allows tests to be created earlier in the development process, and be ready for execution before the program is finished. Additionally, when the tests are generated, the test engineer will often find inconsistencies and ambiguities in the specifications, which allows problems to be found and eliminated early. Specifications can be used as a basis for output checking, which significantly reduces one of the major costs of testing. Another advantage is that the essential part of the test data can be independent of any particular implementation of the specifications. Specification-based testing is also important for conformance testing, where access to the code is not provided, but specifications for the product are.

This paper investigates the usefulness of three recently developed techniques for generating tests from state-based formal specifications: full predicate testing (called *FP testing*) [22, 24], transition-pair testing (called *TP testing*) [22, 24], and specification-mutation testing (called *SM testing*) [2]. The goal of this paper is to identify the relative advantages and disadvantages of each testing method. Comparing testing techniques helps managers decide which to apply, helps researchers and practitioners to improve the techniques, and helps test engineers decide when to apply them.

It is often difficult to compare testing criteria theoreti-

[*]Supported in part by Rockwell Collins, Inc, and in part by the National Science Foundation under grant CCR-98-04111.

[†]Supported in part by the National Institute of Standards and Technology and in part by the National Science Foundation under grant CCR-99-01030.

cally or analytically. Empirical comparisons of testing criteria are also difficult, which is why they are rare, and usually small in scale. While other comparative studies have been carried out for code-based testing techniques [12, 21], we know of no such study for specification-based testing techniques. This paper presents an empirical comparison of FP testing, TP testing, and SM testing. There are two relationships that have been defined elsewhere. Weyuker, Weiss, and Hamlet [25] suggest a relationship called PROBBETTER. A testing criterion C_1 is PROBBETTER than C_2 for a program P if a randomly selected test set T that satisfies C_1 is more "likely" to detect a failure than a randomly selected test set that satisfies C_2. Mathur and Wong [18, 26] suggest a different relationship called PROBSUBSUMES. A testing criterion C_1 PROBSUBSUMES C_2 for a program P if a test set T that is adequate with respect to C_1 is "likely" to be adequate with respect to C_2. If C_1 PROBSUBSUMES C_2, C_1 is said to be more "difficult" to satisfy than C_2.

The PROBBETTER relation is defined with respect to the fault detection capability of test sets, whereas the PROBSUBSUMES relation is defined with respect to the difficulty of satisfying one criterion in terms of another. Both are probabilistic relations between two testing criteria and are defined in terms of specific programs. Although this means that it is difficult to draw general conclusions from any one study, as the number and variety of programs studied increases, our confidence in the validity of a PROBSUBSUMES or a PROBBETTER relationship with a larger set of programs also increases. This paper uses Mathur and Wong's [18] ProbSubsumes measure to compare the three criteria.

The SMV model checker is used to both generate tests and to evaluate test sets with respect to a given criterion. The model checking approach to formal methods specifies a system with a state transition relation and then characterizes the relation with properties that are stated in a temporal logic. Model checking has been successfully applied to a wide variety of practical problems, including hardware design, protocol analysis, operating systems, reactive systems, fault tolerance, and security. Although model checking began as a method for verifying hardware designs, there is growing evidence that it can be applied with considerable automation to specifications for relatively large software systems, such as TCAS II [6].

The three criteria being compared are defined in Section 2. Section 3 presents the empirical method and process, and Section 4 presents the design and conduct of the experiment. Section 5 gives results and analysis, and conclusions and future directions are in Section 6.

2. Test Coverage of State-based Specifications

State-based specifications describe software in terms of states and transitions. Typical state-based specifications define *preconditions* on transitions, which are values that specific variables must have for the transition to be enabled, and *triggering events*, which are changes in variable values that cause the transition to be taken. A triggering event is a change in one or more variable that can cause a transition to be taken, and a new state to be entered. For example, SCR [13, 15] calls these WHEN conditions and triggering events. The values the triggering events have before the transition are sometimes called *before-values*, and the values after the transition are sometimes called *after-values*. The state immediately preceding the transition is the *pre-state*, and the state after the transition is the *post-state*.

The first two criteria, full predicate testing and transition-pair testing, are defined with respect to a state-based specification such as SCR. These have been applied to Flight Guidance System (FGS) [20, 19]. The third criterion, specification-mutation testing, is defined with respect to a model checking specification – in this case, SMV. This criterion has been applied to a subset of FGS. These application show that the criteria can be applied to reasonable large systems. The connection is that state-based specifications such as SCR may be represented as model checking specifications. For example, automated conversions to a model checking representation are available from SCR to SPIN [14] and from StateCharts to SMV [6].

2.1. Full Predicate Coverage

Offutt has defined two coverage criteria, full predicate and transition-pair, in prior research [22, 23, 24]. Full predicate coverage questions whether predicates in the specifications are formulated correctly. Small inaccuracies in the specification predicates can lead to major problems in the software. The full predicate coverage criterion takes the philosophy that to test the software, testers should at minimum provide inputs derived from each clause in each predicate. This criterion requires that each clause in each predicate on each transition is tested independently, thus attempting to address the question of whether each clause is necessary and is formulated correctly.

In the following definitions, T is a set of test cases. Although the tests are intended to be executed on an implementation of the specification, we say that a test *traverses* a transition to indicate that, from a modeling perspective, the test causes the transition's predicate to be true, and the implementation will change from the transition's pre-state to its post-state. Formally:

FP testing:: For each predicate P on each transition, T includes tests that cause each clause c in P to result in a pair of outcomes where the value of P is directly correlated with the value of c.

In this definition, "directly correlated" means that c controls the value of P, that is, one of two situations occurs. Either c and P have the same value (c is true implies P is true and c is false implies P is false), or c and P have opposite values (c is true implies P is false and c is false implies P is true). This explicitly disallows cases such as c is true implies P is true and c is false implies P is true. FP testing is closely related to, but strictly weaker than, the MC/DC (multiple condition/decision coverage) criterion used in testing critical avionics applications [7].

2.2. Transition-pair Coverage

Many mistakes in software can arise because the engineers do not fully understand the complex interactions among sequences of states in the specifications. Full predicate coverage tests transitions independently, but does not test sequences of state transitions, thus some faults may not be adequately tested for. Typical faults that may occur are because an invalid sequence of transitions is allowed, or a valid sequence is not allowed. To check for these kinds of faults, transition-pair coverage requires that pairs of transitions be taken. Transition-pair testing is defined as follows:

TP testing:: For each pair of adjacent transitions $S_i : S_j$ and $S_j : S_k$, T contains a test that traverses the pair of transitions in sequence.

2.3. Specification Mutation Coverage

Mutation testing is a well studied technique for testing software units [10, 11]. It is defined in terms of the code statements in an implementation. Mutation was adapted to the problem of deriving tests from functional specifications by Ammann and Black [1, 3]. A specification for model checking has two parts. One part is a state machine defined in terms of variables, initial values for the variables, and a description of the conditions under which variables may change value. The other part is temporal logic constraints on valid execution paths. Conceptually, a model checker visits all reachable states and verifies that the invariants and temporal logic constraints are satisfied. Model checkers exploit clever ways to avoid brute force exploration of the state space, for example, see Burch et al. [5] or Clarke et al. [8].

Model checkers are attractive for test case generation because given a temporal logic constraint that does not hold, a model checker can (sometimes - see below) produce a counterexample. A counterexample is a *trace*, or sequence of transitions in the state machine. The trace has a direct interpretation as test case, and is useful for not only identifying inputs, but verifying outputs as well.

In the context of the mutation model developed previously [1], Figure 1 illustrates a key difference between mutations to program code and mutations to logic formulae. Code mutants are classified as either equivalent or nonequivalent. An equivalent mutant is one that has exactly the same input/output relation as the original program.

Mutations to logic constraints in a model checking specification result in a different situation. Instead of being either equivalent or nonequivalent, mutants are either *consistent* or *inconsistent* with the state machine. A consistent mutant is a temporal logic formula that is true over all possible traces defined by the state machine. Just as equivalent mutants cannot be distinguished from the original for program-based mutation analysis, consistent mutants cannot be found false for model checking mutation analysis. Fortunately consistency is decidable for these temporal logics, and model checkers are specifically designed to efficiently determine whether or not a temporal logic formula is consistent. So this arena does not have the problem of undecidability or requiring human judgment.

For inconsistent mutants, there are two possibilities. Some temporal logic formulae can be shown to be inconsistent with a single trace through the state machine. For example, if the assertion "the East-West light is never green while the North-South light is green" is inconsistent, the inconsistency could be exhibited with an execution trace that starts in some initial condition and ends in a state where both lights are green. A *falsifiable* mutant is one that can be demonstrated to be inconsistent. Other temporal logic formula may be inconsistent with respect to the state machine, but cannot be shown to be inconsistent with a single trace. For example, an inconsistent assertion that "eventually both the East-South and West-North left turn lights are green simultaneously" cannot be shown to be false with any single trace from the state machine. The reason is that counterexamples are excellent for showing that universally quantified properties fail, but are not useful for showing that existentially quantified properties fail. A *nonfalsifiable* mutant is one that is inconsistent but no counterexample has been found.

For this paper, the specification clauses constraining the transition relation are produced through the processes of expoundment and reflection [1]. *Expoundment* makes implicit aspects of the transition relation explicit, and is essentially the same as making explicit the predicate on the `default` branch of a `case` statement in a traditional programming language. *Reflection* is a process of restating the transition relation in temporal logic. The goal of expoundment and

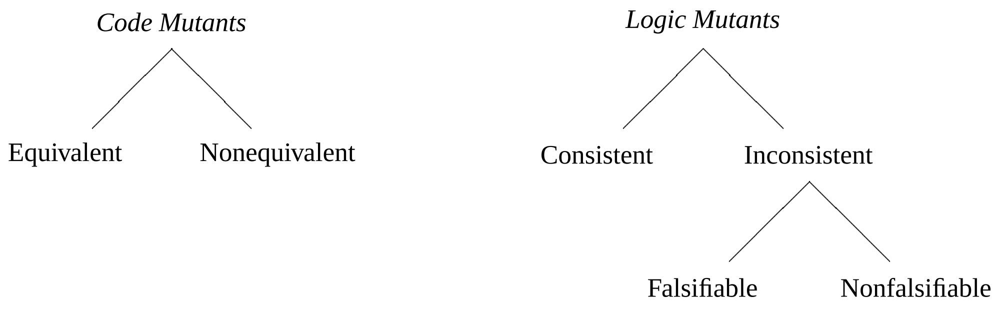


Figure 1. Categories of Mutants.

reflection is to produce a complete, redundant description of the state machine. This allows the model checker to find traces (test cases) that distinguish the behavior of the original description (the state machine) from mutations of an equivalent description (the temporal logic description).

Specification-mutation testing is defined as follows:

SM testing:: For a given transition relation, set of specification clauses constraining the transition relation, and set of mutation operators, let M be the set of mutant specification clauses produced by applying the operators to the specification clauses in all possible ways. Let M' be the inconsistent, falsifiable specification clauses in M. Test set T is mutation adequate with respect to the transition relation, specification clauses, and mutation operators if for each m' in M', some test t in T is a counterexample for m'.

3. Evaluation of Testing Criteria

Analytical comparisons show theoretical relationships between techniques, and are most satisfying because they allow claims that are true in all situations. Empirical comparisons show relations that are based on specific studies. Although it is difficult to show that empirical results hold in all situations, analytical comparisons cannot always be made, but empirical comparisons can.

The PROBSUBSUMES relation used in this paper is defined with respect to the difficulty of satisfying one criterion in terms of another; this has also been called "cross-scoring" [17]. It is a probabilistic relation between two testing criteria and is defined in terms of specific programs. Although this means that it is difficult to draw general conclusions from any one study, as the number and variety of programs studied increases, our confidence in the validity of a PROBSUBSUMES relationship with a larger set of programs also increases.

Mathur and Wong [18] used the PROBSUBSUMES relation in experimental comparisons of all-uses data flow testing with mutation testing, by manually generating test data to satisfy both criteria and comparing the scores. They used four programs and 30 sets of test cases per program and detected equivalent mutants and unexecutable subpaths by hand. This study indicated that mutation-adequate test sets were closer to being data flow-adequate than data flow-adequate test sets were to being mutation-adequate. Offutt et al. [21] also compared all-uses data flow with mutation using more programs, and also compared the two criteria based on the number of faults they found (using the PROBBETTER relationship [25]).

While other comparative studies have been carried out for code-based testing techniques [9, 12, 21], we know of no such study for specification-based testing techniques.

4. Experimental Hypothesis and Conduct

It seems reasonable to suppose that if test sets created for one criterion also satisfy another, then the first criterion can in some sense be considered to be "better" than the second. This is the essence of the PROBSUBSUMES relationship. Thus, this experiment tries to determine if specification mutation adequate test sets are likely to cover full predicate and transition-pair, full predicate adequate test sets are likely to cover transition-pair and specification mutation, and transition-pair adequate test sets are likely to cover full predicate and specification mutation.

For this comparison, the following hypothesis has been formulated for each possible pair from the three criteria: mutation adequacy, full predicate coverage, and transition-pair coverage.

Hypothesis: Criterion i PROBSUBSUMES criterion j.

Since only one example program is evaluated and tests are generated in only one way for that program (albeit automatically way), the hypothesis cannot be completely tested, which is a threat to the external validity of the results. However, this approach does provide the opportunity to discover obvious trends with the aim of spurring further analysis. We limit our results to such a discussion.

The cruise control system is used as our artifact. Cruise control is a common example in the literature [4, 16, 24], and specifications are readily available. The cruise control specification has four modes and 12 mode transitions. The 12 mode transitions form 12 transition predicates. Although the example is very simple, it is sufficient to draw the high level conclusions presented in this paper.

The SCR specification used for the cruise control system is shown in Table 1. The specification has four states: OFF (the initial state), INACTIVE, CRUISE, and OVERRIDE. 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 the table 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). This means that the value of C must change for the transition to be taken, that is, "@T(C)" means C must change from false to true, and "@F(C)" means C must change from true to false. A table entry of `t` or `f` represents a WHEN condition. WHEN[C] means the transition can only be taken if C is true, and WHEN[$\neg$C] 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).

Test sets were generated and evaluated as illustrated in Figure 2. We explain generation of test cases first. The SCR specification for the cruise control example was analyzed by hand to determine the test requirements for FP testing and TP testing. In principle, test cases to satisfy FP testing and TP testing could be constructed directly, but we chose instead to implement test case generation with a model checker. The reasons were (1) to avoid human bias and mistake in generating tests and (2) to use a uniform mechanism to generate and evaluate all tests in the experiment.

To make use of a model checker, a transition relation was developed in SMV for the cruise control problem. This process was performed by hand, although automated tools do exist [4, 14]. At this point in the process, the temporal logic constraints on the transition relation, that is, the `SPEC` clauses in SMV, are empty.

To generate test cases, `SPEC` clauses tailored to the desired criteria were generated and SMV was executed. The set of counterexamples produced by SMV form a test set that is adequate with respect to the relevant criteria. Thus, to generate FP tests, the test requirements for FP were expressed in SMV `SPEC` clauses.[1] Similarly, to generate TP tests, the test requirements for TP testing were expressed in SMV `SPEC` clauses.

For SM testing, the situation is slightly different. The transition relation is expounded and reflected into the temporal logic, and then mutation operators are applied to the temporal logic [1]. Running SMV produces a set of counterexamples that form a mutation adequate test set. The mutation operators used are listed below. The first four are from previous work [1] and the remaining two are new operators. Each operator is illustrated with a mutant it generates from the following clause. Changes are emphasized by underlining.

```
     AG(CruiseControl=Cruise
  -> AX(Toofast ->
  CruiseControl=Inactive))
```

1. *replace_constant* – replace one constant with another.

```
     AG(CruiseControl=Override
  -> AX(Toofast ->
  CruiseControl=Inactive))
```

2. *replace_oper* – replace one operator with another operator, for example, replace "implies" with "or".

```
     AG(CruiseControl=Cruise
  -> AX(Toofast |
  CruiseControl=Inactive))
```

3. *replace_vars* – replace a variable with another variable.

```
     AG(CruiseControl=Cruise
  -> AX(Brake ->
  CruiseControl=Inactive))
```

4. *remove_expr* – remove a simple expression from conjunctions, disjunctions, and implications.

```
     AG(CruiseControl=Cruise
  -> AX(__
  CruiseControl=Inactive))
```

5. *negate_expr* – replace a simple expression with its negation.

```
     AG(CruiseControl=Cruise
  -> AX(!Toofast ->
  CruiseControl=Inactive))
```

6. *constant_expr* – replace a simple expression with the constant $TRUE$. Repeat with the constant $FALSE$. This mutation operator mimics the "stuck-at" faults from circuit testing and is quite powerful.

[1] The general strategy is to write a temporal logic formula that expresses the *negation* of a given test requirement. If the test requirement is satisfiable, the model checker produces a counterexample to the negated requirement; the counterexample is a test case that satisfies the test requirement. Of course, if the test requirement is not satisfiable, no counterexample, and hence no test case, exists for that requirement.

Previous Mode	Ignited	EngRun	Toofast	Brake	Enum1	New Mode
Off	@T	-	-	-	-	Inactive
Inactive	@F	-	-	-	-	Off
	-	t	f	f	@T(Enum1=Activate)	Cruise
Cruise	@F	-	-	-	-	Off
	-	@F	-	-	-	Inactive
	-	-	@T	-	-	
	-	-	-	@T	-	Override
	-	-	-	-	@T(Enum1=Deactivate)	
Override	@F	-	-	-	-	Off
	-	@F	-	-	-	Inactive
	-	-	f	f	@T(Enum1=Activate)	Cruise
	-	-	f	f	@T(Enum1=Resume)	

Table 1. SCR Specifications for the Cruise Control System.

```
   AG(CruiseControl=Cruise
-> AX(TRUE ->
CruiseControl=Inactive))
```

At this point in the process, test sets adequate for each of the three criteria have been generated; this corresponds to the right hand side of the Test Case Generation diagram in Figure 2.

We now turn to evaluation of one criterion against another. Specifically, we wish to know how the test set generated for one criterion scores against the test requirements for another criterion. This is the PROBSUBSUMES relation defined in Section 3. This was done using the coverage tools developed previously [1]. In general terms, each test case from the test set under evaluation is turned into a "forced" state machine. SMV is used to see which `SPEC` clauses from the criteria being evaluated are found inconsistent. The results for the entire test set are collected and evaluated. The score for a test set is the number of satisfied test requirements relative to the number of total test requirements. This process is shown in the Test Case Evaluation diagram in Figure 2.

5. Results and Analysis

The three techniques of full predicate, transition-pair, and specification mutation testing are compared on two bases, a coverage measurement (using the PROBSUBSUMES relationship) and test set size.

Coverage is defined in the usual way, as the percentage of test requirements covered. The coverage for a set of tests that are adequate with respect to criterion A are measured on a second criterion B. Thus coverage of criterion A by criterion B is 100% if and only if a test set that is adequate for criterion A is also adequate for criterion B. More formally, let A and B be two adequacy criteria, and $F_A\ (T, S)$ and $F_B\ (T, S)$ be the functions that measure whether a test set T is adequate for the criteria. Let T_A be a set of test data that is adequate with respect to criterion A and T_B be a set of test data that is adequate with respect to criterion B. Then the coverage of criterion A by criterion B is $F_A\ (T_B, S)$ and the coverage of criterion B by criterion A is $F_B\ (T_A, S)$. Since a criterion covers itself, by definition $F_A\ (T_A, S) = 100\%$ and $F_B\ (T_B, S) = 100\%$.

If F_t is the total number of full predicate requirements for the specification being tested, F_s is the number of full predicate requirements that have been satisfied by the test set and F_i is the number of full predicate requirements that can never be satisfied because of infeasible predicates, then the *full predicate score* is:

$$FPS\,(S, T) = \frac{F_s}{F_t - F_i}. \tag{1}$$

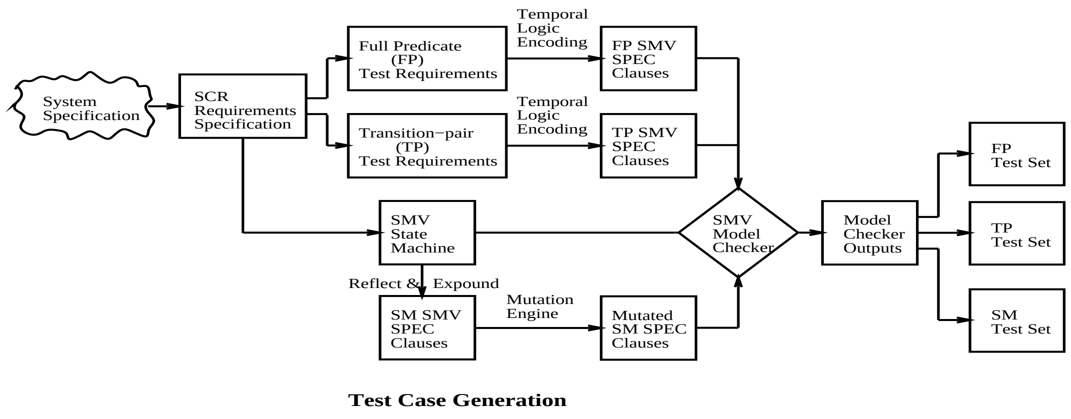
If T_t is the total number of transition-pair requirements for the specification being tested, T_s is the number of transition-pair requirements that have been satisfied by the test set and T_i is the number of transition-pair requirements that can never be satisfied because of infeasible pairs of transitions, then the transition-pair score is:

$$TPS\,(S, T) = \frac{T_s}{T_t - T_i}. \tag{2}$$

The coverage measure for mutation is the *mutation score*. If M_t is the total number of mutants generated for a specification, M_k is the number of mutants killed by a set of test cases T, and M_c is the number of consistent mutants for the specification being tested, then the mutation score is:

$$SMS\,(S, T) = \frac{M_k}{M_t - M_c}. \tag{3}$$

In this experiment, if the mutation criterion is denoted by M, the full predicate criterion is denoted by F, and the transition-pair criterion is denoted by T, then F_M computes the mutation score for a set of test data using Equation 3, F_P computes the full predicate score for a set of test data using Equation 1, and F_T computes the transition-pair score for a set of test data using Equation 2. Values of $F_M\ (T_F, S)$, $F_M\ (T_T, S)$, $F_F\ (T_M, S)$, $F_F\ (T_T, S)$, $F_T\ (T_M, S)$ and



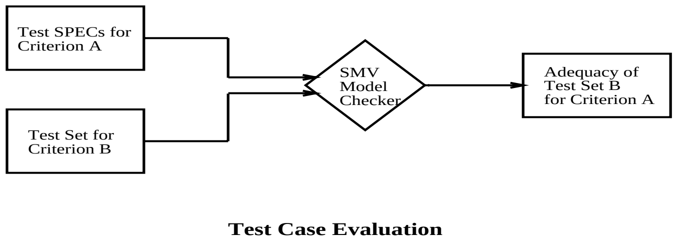

Test SPECs for Criterion A
Test Set for Criterion B
SMV Model Checker
Adequacy of Test Set B for Criterion A
Test Case Evaluation

Figure 2. Experimental Process.

Test Case Set	Full Predicate Score	Transition-pair Score	Mutation Score
Full Predicate TC	100	32	86
Transition-pair TC	50	100	82
Mutation TC	88	38	100

Table 2. Coverage Scores of Test Criteria.

$F_T\ (T_F, S)$ were computed for the test case sets generated from the cruise control specification. These scores are shown in Table 2.

Table 2 shows that the FP-tests (left column) were 100% adequate for FP-testing (second column), only 32% adequate for TP-testing (third column), and 86% adequate for SM-testing (second column). Both the FP-tests and the SM-tests scored very high on each other's criterion, but neither scored very highly on the transition-pair criterion. However, the transition-pair tests did not score very high on the full predicate criterion, but scored relatively highly on the specification mutation criterion. This indicates that full predicate testing and specification mutation testing might be similar in effectiveness, and possibly even interchangeable, but that transition-pair testing may yield very different tests. We are currently investigating the theoretical relation between the FP and SM criteria. The reflection process for mutation analysis is not deterministic – there are a variety of logically equivalent temporal logic formulae that are nonetheless different syntactically and therefore lead to a different test cases. The authors conjecture that a thorough understanding of the reflection process would lead to a set of test cases from mutation analysis that would, in fact, be FP adequate.

In terms of the six experimental hypotheses, no pair of criteria appeared to satisfy the PROBSUBSUMES relation. However, given the closeness of FP and SM coverage results, it may be that a PROBSUBSUMES hypothesis is satisfied for modified versions of one or both of these criteria. We intend to pursue research to investigate this possibility.

5.1. Test Set Size

Table 3 gives the number of test cases generated for each criterion. In the cruise control software, specification mutation required about twice as many tests as full predicate and transition-pair. The size of the full predicate test set is linearly bound by the number of clauses in the transition predicates, and the size of the transition-pair test set is bounded by the square of the number of transitions. The size of the mutation test set is harder to bound. In general, the number of mutations is limited by the size of the *description* of the state machine and the set of mutation operations chosen. It is important to note that the number of mutations does not "blow up" as does the state space, which is a common problem with model checking.

Test Case Set	**Test Set Size**
Full Predicate TC	21
Transition-pair TC	24
Mutation TC	42

Table 3. Number of Test Cases Per Criterion.

6. Conclusions

This paper has presented results from an empirical study of three specification-based test generation criteria. First, full predicate coverage, transition-pair coverage, and statement mutation were compared on the basis of a "cross scoring", where tests generated for each criterion are measured against the other. Second, the three techniques were compared on the basis of the number of test cases generated to satisfy them, in a rough attempt to compare their relative costs.

For the program used, the specification mutation score of the full predicate tests and the full predicate scores of the specification mutation tests were quite high. This provides some evidence that specification mutation PROBSUBSUMES full predicate and full predicate PROBSUBSUMES specification mutation, that is, the two techniques are relatively similar. However, neither the full predicate tests nor the specification mutation tests had high transition-pair scores, and the transition-pair tests did not have high full predicate or specification mutation scores. Thus, it can be inferred that transition-pair tests offer something different from full predicate and specification mutation tests.

Of course, this experiment has limitations that are difficult to avoid in this area. To do certain kinds of statistical analyses, we must be able to assume that the data is based on "representative samples" from the population being studied. Unfortunately, there is currently no way to choose a representative sample of software, test cases, or faults, so we are limited in our ability to use statistical analysis tools to make claims of significance.

The program studied was relatively small, which leaves the question of how the conclusions might scale up to large software systems. Experimentation is currently underway to apply these techniques to a larger industrial system.

References

[1] Paul E. Ammann and Paul E. Black. A specification-based coverage metric to evaluate test sets. In *HASE 99: Fourth IEEE International Symposium on High Assurance Systems*, pages 239–248, Washington, DC, November 1999.

[2] 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.

[3] Paul E. Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. In *Second IEEE International Conference on Formal Engineering Methods*, pages 46–54, Brisbane, Australia, 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] J. Burch, E. Clarke, K. McMillan, D. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Annual Symposium on Logic in Computer Science*, pages 428–439, June 1990.

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

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

[8] E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, Cambridge, Massachusetts, USA, 2000.

[9] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A comparison of data flow path selection criteria. In *Proceedings of the Eighth International Conference on Software Engineering*, pages 244–251, London UK, August 1985. IEEE Computer Society Press.

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

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

[12] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, August 1993.

[13] C. Heitmeyer, R. Jefords, and B. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering Methodology*, 5(3):231–261, July 1996.

[14] Constance L. Heitmeyer, B. Labaw, and D. Kiskis. Consistency checking of SCR-style requirements specifications. In *Proceedings, International Symposium on Requirements Engineering*, York, UK, March 1995.

[15] K. Henninger. Specifiying software requirements for complex systems: New techniques and their applications. *IEEE Transactions on Software Engineering*, SE-6(1):2–12, January 1980.

[16] Zhenyi Jin. Deriving mode invariants from SCR specifications. In *Proceedings of Second IEEE International Conference on Engineering of Complex Computer Systems*, pages 514–521, Montreal, Canada, October 1996. IEEE Computer Society.

[17] A. P. Mathur. On the relative strengths of data flow and mutation based test adequacy criteria. In *Proceedings of the Sixth Annual Pacific Northwest Software Quality Conference*, Portland OR, 1991. Lawrence and Craig.

[18] A. P. Mathur and W. E. Wong. An empirical comparison of data flow and mutation-based test adequacy criteria. *The Journal of Software Testing, Verification, and Reliability*, 4(1):9–31, March 1994.

[19] S. P. Miller. Specifying the mode logic of a flight guidance system in CoRE and SCR. In *Second Workshop on Formal Methods in Software Practice*, Clearwater Beach, FL, March 1998.

[20] S. P. Miller and K. F. Hoech. Specifying the mode logic of a flight guidance system in CoRE. Technical Report WP97-2011, Rockwell Collins, November 1997.

[21] A. J. Offutt, Jie Pan, Kanupriya Tewary, and Tong Zhang. An experimental evaluation of data flow and mutation testing. *Software–Practice and Experience*, 26(2):165–176, February 1996.

[22] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In *Proceedings of the Second IEEE International Conference on the Unified Modeling Language (UML99)*, pages 416–429, Fort Collins, CO, October 1999. IEEE Computer Society Press.

[23] Jeff Offutt and Shaoying Liu. Generating test data from SOFL specifications. *The Journal of Systems and Software*, 49(1):49–62, December 1999.

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

[25] E. J. Weyuker, S. N. Weiss, and R. G. Hamlet. Comparison of program testing strategies. In *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification*, pages 1–10, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society Press.

[26] Weichen Eric Wong. *On Mutation and Data Flow*. PhD thesis, Purdue University, December 1993. (Also Technical Report SERC-TR-149-P, Software Engineering Research Center, Purdue University, West Lafayette, IN).

[27] 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.