

# Criteria for Generating Specification-based Tests <sup>\*</sup>

A. Jefferson Offutt and Yiwei Xiong  
*Information and Software Engineering*  
*George Mason University*  
*Fairfax, VA 22030-4444 USA*  
*{ofut,yxiong}@ise.gmu.edu*

Shaoying Liu  
*Faculty of Information Sciences*  
*Hiroshima City University*  
*Asaminami-ku, Hiroshima 731-31 Japan*  
*shaoying@cs.hiroshima-cu.ac.jp*

## Abstract

*This paper presents general criteria for generating test inputs from state-based specifications. Software testing can only be formalized and quantified when a solid basis for test generation can be defined. Formal specifications of complex systems represent a significant opportunity for testing because they precisely describe what functions the software is supposed to provide in a form that can easily be manipulated. These techniques provide coverage criteria that are based on the specifications, and are made up of several parts, including test prefixes that contain inputs necessary to put the software into the appropriate state for the test values. The test generation process includes several steps for transforming specifications to tests. Empirical results from a comparative case study application of these criteria are presented.*

**Keywords:** Formal Methods, Specification-based Testing, Software Testing.

## 1 Introduction

There is an increasing need for effective testing of software for complex safety-critical applications, such as avionics, medical, and other control systems. These software systems usually have clear high level descriptions, sometimes in formal representations. Unfortunately, most system level testing techniques are only described informally. This paper is part of a project that is attempting to provide a solid foundation for generating tests from system level software specifications via new coverage criteria. Formal coverage criteria offer testers ways to decide what test inputs to use during testing, making it more likely that the

testers will find any faults in the software and providing greater assurance that the software is of high quality and reliability. Such criteria also provide stopping rules and repeatability. The eventual goal of this project is a general model for generating tests from formal specifications; this paper presents results for generating tests from one kind of formal specifications. Formal specifications represent a significant opportunity for testing because they provide precise descriptions of what functions the software is supposed to provide in a form that can easily be manipulated by automated means.

This paper presents a model for developing test inputs from state-based specifications, and formal criteria for test case selection. Related techniques for developing test inputs that are derived from SOFL specifications [26] are presented elsewhere [30]. The test case generation model includes techniques for generating tests at several levels of detail. These techniques provide coverage criteria that are based on the specifications, and the test generation process details steps for transforming functional specifications to tests.

A common source for tests is the program code. In *code-based test generation*, a testing criterion is imposed on the software to produce test requirements. For example, if the criterion of branch testing is used, the tests are required to cover each branch in the program. The specification  $S$  (which can be formal or informal) is used as a basis for writing the program  $P$ , which is used to generate the tests  $T$ , according to some coverage criterion such as branch or data flow. Execution of  $T$  on  $P$  creates the *actual output*, which must be compared with the *expected output*. The expected output is produced with some knowledge of the specifications. Thus, code-based generation uses the specifications to generate the code and check the output of the tests.

This is in contrast to *specification-based testing*, in which specifications are used to produce tests, as well as to produce the program. One significance of pro-

---

<sup>\*</sup>This work is supported in part by Rockwell Collins, Inc, in part by the Ministry of Education of Japan under Grant-in-Aid for Scientific Research on Priority Areas (A) (No. 09245105), Grant-in-Aid for Scientific Research (B) (No. 11694173) and (C) (No. 11680368), and in part by the U.S. National Science Foundation under grant CCR-98-04111.

ducing tests from specifications is that the tests can 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, allowing the specifications to be improved before the program is written. This project is looking at ways to generate tests from specifications; others, such as Li et al., have been developing techniques for creating expected output from specifications [25, 20, 28].

Specification-based test generation has several advantages, particularly when compared to code-based generation. Requirements/specifications can be used as a basis for output checking, significantly reducing one of the major costs of testing. The process of generating tests from the specifications will often help the test engineer discover problems with the specifications themselves; if this step is done early, the problems can be eliminated early, saving time and resources. Generating tests during development also allows testing activities to be shifted to an earlier part of the development process, allowing for more effective planning and utilization of resources. Another advantage is that the essential part of the tests can be independent of any particular implementation of the specifications.

Software functional specifications have been incorporated into testing in several ways. They have been used as a basis for test generation, to check the output of software on test inputs [28], and as a basis for formalizing *test specifications* (as opposed to functional specifications) [35]. This paper is primarily concerned with the first use, that of generating tests from specifications, which is referred to as specification-based testing. An immediate goal is to develop *mechanical* procedures to derive tests from formal specifications; long term goals include automated tool support to transform formal functional specifications into effective tests.

Specification-based testing is currently immature, which means there is a scarcity of formalizable criteria and automated tool support. It is this problem that this research is attempting to address. System level testing has the potential to benefit from formal specifications, by using the formal specifications as input to formalizable, automatable test generation processes. Another advantage of specification-based testing is that it can support the automation of test result analysis, by using specifications as test oracles.

The two approaches are sometimes used in combination. The most common approach in industry is to generate tests based on the specifications, and then use *code-based coverage analysis* to measure the qual-

ity of the tests. For example, the tests might be measured by how many branches in the software are covered. No results have been published concerning how effective this combination is. It is known, however, that it is difficult to construct system and subsystem level tests that cover detailed code-level requirements (such as branches). This is why code-based test generation is often thought of as useful for *unit testing*, when individual functions or modules are tested, and specification-based test generation is often thought of as useful for *system testing*, when entire working systems are tested. These are really orthogonal issues, however. Specification-based testing techniques can be and are used at the unit level. The key difference is in the questions that the two approaches attempt to answer. Specification-based testing addresses the question of “why am I testing?”, whereas code-based testing addresses the question of “how much software is being covered during testing?”.

This paper first reviews some of the work in specification-based testing, then presents four related form criteria for generating tests from formal functional specifications. Then a small case study of these criteria is discussed, and finally conclusions and current and future work are presented.

## 2 Approaches to Specification-based Testing

The research literature reports on specific tools for specific formal specification languages [5, 7, 15, 32, 36, 37], manual methods for deriving tests from specifications [1, 12, 16], case studies on using specifications to check the output of the software on specifications [13, 23, 24, 35, 20, 25, 28], and formalizations of test specifications [35, 4, 10]. This paper uses the term *specification-based testing* in the narrow sense of using specifications as a basis for deciding what tests to run on software.

Model-based specification languages, such as Z and VDM, attempt to derive formal specifications of the software based on mathematical models. Spence and Meudec [34] and Dick and Faivre [12] suggested using specifications to produce predicates, and then using predicate satisfaction techniques to generate tests. Stocks and Carrington [35] and Ammann and Offutt [1] proposed using a form of *domain partitioning* to generate tests. Hierons [19] presents algorithms that rewrite Z specifications into a form that can be used to partition the input domain. Hayes [16] has suggested a dynamic scheme that uses *run-time verification* of the program.

Chang and Richardson [8] presented techniques to

derive test conditions from ADL specifications, a predicate logic-based language that is used to describe the relationships between inputs and outputs of program units.

Algebraic specification languages describe software by making formal statements, called *axioms*, about relationships among operations and the functions that operate on them. Gannon, McMullin and Hamlet [15] used a *script derivation* approach to generate strings as tests. Doong and Frankl [13] used a similar approach to test object-oriented software. Bougé et al. [7] suggested a logic programming approach to generating tests from algebraic specifications. Tsai, Volovik, and Keefe [36] used a similar approach, but started with relational algebra queries.

State-based specifications describe software in terms of state 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 trigger event “triggers” the change in state. For example, SCR [18, 3] 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 *post-state* is the state after the transition.

Blackburn [6] used state-based functional specifications of the software, expressed in the language **T-Vec**, to derive disjunctive normal form constraints, which are solved to generate tests. Weyuker, Goradia, and Singh [37] present a method to generate tests from boolean logic specifications of software.

This work is also related to work in the area of protocol conformance testing. For example, Luo et al. [27] have presented a method for generating test sequences from finite-state machines for testing communication protocols. The problem is very similar to the one addressed in this paper, and the methods are related, but the finite-state machine model used for conformance testing is more narrow than state-based functional specifications for software. Moreover, the problems of protocol conformance testing are more restricted and can be more completely defined than the problems of fault detection in general software.

Most of the current specification-based testing techniques use manual methods that cannot be easily generalized or automated. Goals of the current research include generalizing the currently known techniques, defining measurable criteria, and developing automated tools.

### 3 Specification-based Testing Criteria

An important problem in software testing is deciding when to stop. A *test case* is a collection of inputs that cause one execution of a software program. Test cases are run on software to find failures and gain some confidence in the software. Unfortunately, the entire domain of the software (which in most cases is effectively infinite) cannot be exhaustively searched. Adequacy criteria are therefore defined for testers to decide whether software has been adequately tested for a specific testing criterion [14].

*Test requirements* are specific things that must be satisfied or covered, for example, reaching statements are the requirements for statement coverage, killing mutants are the requirements for mutation, and executing DU pairs are the requirements in data flow testing. A *testing criterion* is a rule or collection of rules that impose requirements on a set of test cases. Test engineers measure the extent to which a criterion is satisfied in terms of *coverage*, which is the percent of requirements that are satisfied.

This paper introduces several criteria for system level testing. These criteria are expected to be used both to guide the testers during system testing and to help the testers find rational, mathematical-based points at which to stop testing.

In these criteria, tests are generated as multi-part, multi-step, multi-level artifacts. The multi-part aspect means that a test case is composed of several components: *test case values*, *prefix values*, *verify values*, *exit commands*, and *expected outputs*. Test case values directly satisfy the test requirements, and the other components supply supporting values. The multi-step aspect means that tests are generated in several steps from the functional specifications by a refinement process. The functional specifications are first refined into test specifications, which are then refined into test scripts. The multi-level aspect means that tests are generated to test the software at several levels of abstraction.

A *test case value* is the essential part of a test case, the values that come from the test requirements. It may be a command, user inputs, or a software function and values for its parameters. In state-based software, test case values are usually derived directly from triggering events and preconditions for transitions. A test case *prefix value* includes all inputs necessary to reach the pre-state and to give the triggering event variables their before-values. Any inputs that are necessary to show the results are *verify values*, and *exit commands* depend on the system being tested. *Expected outputs* are created from the after-values of the

triggering events and any postconditions that are associated with the transition.

This paper presents four different test criteria, each of which requires a different amount of testing: (1) the transition coverage criterion, (2) the full predicate coverage criterion, (3) the transition-pair coverage criterion, and (4) the complete sequence criterion. These are defined in the next four subsections. These criteria are defined in terms of a state-based requirement/specification. These criteria assume that the software specifications are described as a number of *states* that the software may be in, and *transitions* among these states. Transitions define under what conditions the software will change states, and are given as mathematical expressions involving variables and arithmetic, relational, and binary operations.

To apply the criteria, a state-based requirement/specification is represented as a directed graph, called the *specification graph*. Two of these criteria are similar to traditional code-level structural coverage criteria, except the criteria are defined on the specification graph rather than a control flow graph. Transition coverage is analogous to branch coverage. Full predicate coverage relies on definitions from Do178B [33], and the definition is similar to that of modified condition/decision coverage (MC/DC) [9], which requires that every decision and every condition within the decision has taken every outcome at least once, and every condition has been shown to independently affect its decision.

It is possible to apply all criteria, or to choose a criterion based on a cost/benefit tradeoff. The first two are related; the transition coverage criterion requires many fewer tests than the full predicate coverage criterion, but if the full predicate coverage criterion is used, the tests will also satisfy the transition coverage criterion (full predicate coverage subsumes transition coverage). Thus only one of these two should be used. The latter two criteria are meant to be independent; transition-pair coverage is intended to check the interfaces among states, and complete sequence testing is intended to check the software by executing the software through complete execution paths. As it happens, transition-pair coverage subsumes transition coverage, but they are designed to test the software in very different ways.

### 3.1 Transition coverage criterion

It is felt that at a minimum, a tester should test every precondition in the specification at least once. This philosophy is defined in terms of the specification graph by requiring that each transition is taken. In the

criteria definitions,  $T$  is a set of test cases, and  $SG$  is a specification graph.

**Transition coverage:  
the test set  $T$  must satisfy every transition in the  $SG$ .**

### 3.2 Full predicate coverage criterion

One question during testing is whether the 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. Assuming the Boolean operators are AND, OR, and NOT, Boolean expression, clause and predicate are defined as follows:

- A *Boolean expression* is an expression whose value can be either **True** or **False**.
- A *clause* is a Boolean expression that contains no Boolean operators. For example, relational expressions and Boolean variables are clauses. (“Clause” is the term typically used in math texts, Do178B [33] uses the term “conditions”.)
- A *predicate* is a Boolean expression that is composed of clauses and zero or more Boolean operators. A predicate without a Boolean operator is also a clause. If a clause appears more than once in a predicate, each occurrence is a distinct clause.

Full predicate coverage is based on the philosophy that each clause should be tested independently, that is, while not being influenced by the other clauses. In other words, each clause in each predicate on every transition must independently affect the value of the predicate. 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.

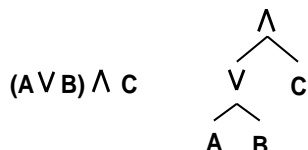
**Full predicate coverage:**  
**for each predicate  $P$  on each transition,  $T$  must include tests that cause each clause  $c$  in  $P$  to result in a pair of outcomes where the value of  $P$  is directly correlated with the value of  $c$ .**

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.

Note that if full predicate coverage is achieved, transition coverage will also be achieved. To satisfy the requirement that the *test clause* controls the value of the predicate, other clauses must have specific values. If the predicate is  $(X \wedge Y)$  and the test clause is  $X$ ,  $Y$  must be **True**. Likewise, if the predicate is  $(X \vee Y)$ ,  $Y$  must be **False**.

### 3.2.1 Satisfying full predicate coverage

Although there are several ways to find values that satisfy full predicate coverage, the simplest way is to use an expression parse tree. An expression parse tree is a binary tree that has binary and unary operators for internal nodes, and variables and constants at leaf nodes. The relevant binary operators are **and** ( $\wedge$ ) and **or** ( $\vee$ ); the relevant unary operator is **not**. For example, the expression parse tree for  $(A \vee B) \wedge C$  is:



Given a parse tree, full predicate coverage is satisfied by walking the tree. First, a test clause is chosen. Then the parse tree is walked from the test clause up to the root, then from the root down to each clause. While walking up a tree if a given clause’s parent is **or**, its sibling must have the value of **False**. If its parent is **and**, its sibling must have the value of **True**. If a node is the inverse operator **not**, the parent node is given the inverse value of the child node. This is repeated for each node between the test clause and the root.

Once the root is reached, values are propagated down the unmarked subtrees using a simple tree walk.

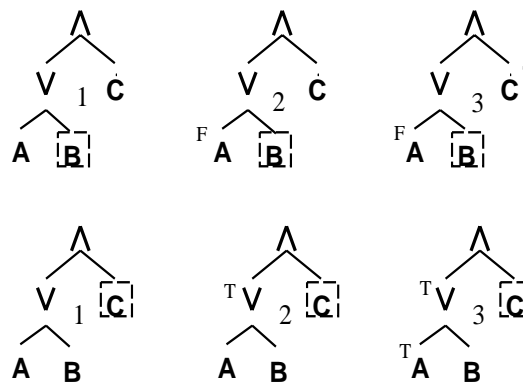


Figure 1: Constructing test case requirements from an expression parse tree

If an **and** node has the value of **True**, then both children must have the value **True**; if an **and** node has the value of **False**, then either child must have the value **False** (which one is arbitrary). If an **or** node has the value of **False**, then both children must have the value **False**; if an **or** node has the value of **True**, then either child must have the value **True** (which one is arbitrary). If a node is the inverse operator **not**, the child node is given the inverse value of the parent node.

Figure 1 illustrates the process for the expression above, showing both  $B$  and  $C$  as test clauses. In the top sequence,  $B$  is the test clause (shown with a dashed box). In tree 2, its sibling,  $A$ , is assigned the value **False**, and in tree 3,  $C$  is assigned the value **True**. In the bottom sequence,  $C$  is the test clause. In tree 2,  $C$ ’s sibling is an **or** node, and is assigned the value **True**. In tree 3,  $A$  is assigned the value **True**. Note that in tree 3, either  $A$  or  $B$  could be given the **True** value; the choice is arbitrary.

These test cases sample from both valid and invalid transitions, with only one transition being valid at a time. Invalid transitions are tested by violating the appropriate preconditions. In addition, the test engineer may choose semantically meaningful combinations of conditions. Testing with invalid inputs can help find faults in the implementation as well as the formulation of the specifications.

As a concrete example, consider the formula whose parse tree was given above,  $(A \vee B) \wedge C$ . The following partial truth table provides the values for the test clauses in bold face. To ensure the requirement that the test clause must control the final result, the partial truth table must be filled out as follows (for the last two entries, either  $A$  or  $B$  could have been **True**, both were assigned the value **True**):

	$(A \vee B)$	$\wedge$	$C$
1	<b>T</b>	F	<b>T</b>
2	<b>F</b>	F	F
3	F	<b>T</b>	<b>T</b>
4	F	<b>F</b>	F
5	T	T	<b>T</b>
6	T	T	<b>F</b>

### 3.2.2 Handling triggering events

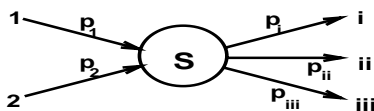
As defined in Section 2, a triggering event is a change in a value for a variable, expression, or expressions that causes the software to transition from one state to another. In SCR and CoRE, triggering event variables have a different format from other variables in transition predicates. A triggering event actually specifies two values, a before-value and an after-value. To fully test predicates with triggering events, test engineers must distinguish between them by controlling values for both before-values and after-values. This paper suggests implementing this by assuming two versions of the triggering event variable,  $A$  and  $A'$ , where  $A$  represents the before-value of  $A$  and  $A'$  represents its after-value.

### 3.3 Transition-pair coverage criterion

Many mistakes in software can arise because the engineers do not fully understand the complex interactions among sequences of states in the specifications. The previous criteria test transitions independently, but do 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 coverage:**  
**for each pair of adjacent transitions  $S_i : S_j$  and  $S_j : S_k$  in SG, T must contain a test that traverses the pair of transitions in sequence.**

Consider the following state:



To test the state  $S$  at the transition-pair criterion, six tests are required: (1) from 1 to i, (2) 2 to i, (3) 1 to ii, (4) 2 to ii, (5) 1 to iii, and (6) 2 to iii. These tests require inputs that satisfy the following pairs of predicates:  $(P_1:P_i)$ ,  $(P_1:P_{ii})$ ,  $(P_1:P_{iii})$ ,  $(P_2:P_i)$ ,  $(P_2:P_{ii})$ , and  $(P_2:P_{iii})$ .

### 3.4 Complete sequence criterion

It seems very unlikely that any successful test method could be based on purely mechanical methods; at some point the experience and knowledge of the test engineer must be used. Particularly at the system level, effective testing probably requires detailed domain knowledge. A *complete sequence* is a sequence of state transitions that form a complete practical use of the system. In most realistic applications, the number of possible sequences is too large to choose all complete sequences. In many cases, the number of complete sequences is infinite.

**Complete sequence:**  
**T must contain tests that traverse “meaningful sequences” of transitions on the SG, where these sequences are chosen by the test engineer based on experience, domain knowledge, and other human-based knowledge.**

Which sequences to choose is something that can only be determined by the test engineer with the use of domain knowledge and experience. This is the least automatable level of testing.

### 3.5 Summary

This section has introduced four criteria to guide the testers during system level testing. While these criteria are black-box in nature, and depend only on the specifications, not the implementation, they are partly motivated by structural coverage test criteria. It should be emphasized, however, that the notion of coverage is based on the specifications, and there is no guarantee of code coverage. These criteria are designed to provide a range of test strengths, and it is hoped that this will provide a practical range of cost/benefit choices for testers.

### 3.6 Automation notes

It is possible to automate most of the test generation for these criteria. If a machine-readable form of the specification table is available, the transition conditions can be read directly from the table. The

specification graph can then be automatically created from the states and transition conditions. Test requirements take the form of partial truth tables defined on transition predicates, state transition predicates, and pairs of state transition predicates. Given a formal functional specification, most if not all of these test requirements can be generated automatically. The prefix of a test case includes inputs necessary to put the system into a particular pre-state. Given the specification graph, many, if not most, of these prefixes can be generated automatically. One open question is whether this problem is generally solvable (unlike the related reachability problem in general software, which is generally unsolvable), and how to solve or partially solve the problem. It is also possible to automatically refine test specifications into test scripts. Finally, algorithms for automatically generating test scripts can be developed, although the input syntax of the program will be needed. The final step, generating complete sequence tests, cannot be fully automated. But an appropriate interface could present the specification graph, and allow the tester to choose sequences of states by pointing and clicking on the screen. Each time a state is chosen, the transition from the previous state could be automatically translated into values and saved as part of the test case. This would allow the tester's job to become the purely intellectual exercise of choosing sequences of states to be entered.

An automated tool requires that certain restrictions be put on the transition predicates. Solving arbitrary mathematical equations is an undecidable problem, thus the equations must be limited in some way. The simplest limitation would be to require that arithmetic expressions be linear, although this is somewhat more restrictive than necessary. An implementation also must require that there be a way to relate variables that appear in the specifications to program variables. The simplest way is to assume that the names are the same, but a more general approach would be to require a mapping function from specification variables to program variables. A third issue is with duplicate variables. If a variable appears more than once in the same transition predicate, it is possible to have test requirements that impose infeasible combinations of values on the variable (for example, a boolean variable could be required to be both true and false). A separate paper [31] describes a preliminary proof-of-concept tool that automatically generates test data according to these criteria from UML statecharts. This tool currently assumes that expressions are linear, specification variables have the same name as program variables, and that variables appear at most once in each transition predicate.

## 4 Case Study

An empirical study has been undertaken to demonstrate the feasibility of these criteria. The goal was to demonstrate that the specification-based criteria can be effectively used; it is hoped to evaluate them more fully in the future.

Two measurements of the criteria have been carried out. Tests were created and then measured on the basis of the structural coverage criterion of decision testing, and then the tests were measured in terms of their fault-detection abilities. One moderate size program was used (Cruise control [2, 22]), representative faults were seeded, and tests were generated by hand.

Cruise control is a common example in the literature [2, 22], and specifications are readily available. The specifications for a version of the system (note that it does not model the throttle) 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 (*Running*), 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*. SCR specifications for Cruise are given in Table 1.

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. Entries of @T and @F indicates that the value for the condition C must change for the transition to be taken, "@T(C)" means C must change from false to true, and "@F(C)" from true to false. Entries of t are f are WHEN conditions; WHEN[C] means the transition can only be taken if C is true, and WHEN[¬C] if C is false. If a condition C does not affect a transition, the entry contains a hyphen "-" (don't care condition).

To avoid bias, tests were created independently from the faults, by different people. The tests were created manually before any execution. Each test case was executed against each buggy version of **Cruise**. After each execution, failures (if any) were identified.

A model of the cruise control problem was implemented in about 400 lines of C. Cruise has seven functions, 184 blocks, and 174 decisions. Twenty-five faults were created by hand and each was inserted into a separate version of the program. Most of these faults are based on mutation-style modifications, and most were in the logic that implemented the state machine. Four were naturally occurring faults, made during initial implementation.

Table 1: SCR specifications for the cruise control system

Previous Mode	Ignited	Running	Toofast	Brake	Activate	Deactivate	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	-

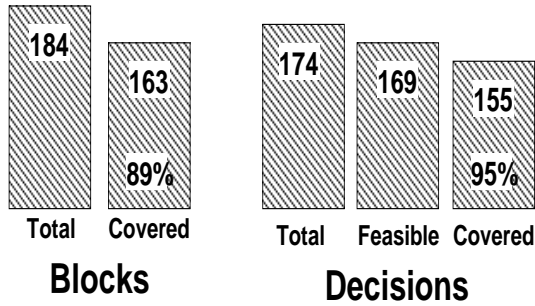


Figure 2: Branch and decision coverage results

#### 4.1 Results and analysis

Block and decision coverage was computed using the full predicate tests. The coverage was measured using Atac [21]. Of the 174 decisions, 5 are infeasible, leaving 169. The results are shown in Figure 2. The 54 test cases covered 163 of the blocks (89%) and 155 of the decisions (95%). Of the 19 uncovered decisions, five were infeasible, and eleven were related to input parameters that were not used during testing. That is, these eleven decisions were not related to the functional specifications. The remaining three decisions were left uncovered because the variables *Activate*, *Deactivate*, and *Resume* are only used as triggering events in the specifications, not condition variables. Thus, there are statements in the software that handle assignments to these variables as WHEN conditions that are never executed. Although there have been very few published studies on the ability of specification-based tests to satisfy code-based coverage criteria, these results seem very promising.

The other measurement was for the fault-detection ability of the tests. Twelve test cases were generated for the transition coverage criterion, and an additional forty-two for the full predicate criterion (making 54 total). As a control comparison, 54 additional test

cases were generated randomly. Although 25 versions of Cruise were created, each one containing one fault, one was such that the program goes into an infinite loop on any input. Since this fault was so trivial, it was discarded. Results from the three sets of tests are shown in Table 2.

Detailed analysis of the faults showed that three of the four faults that the full predicate tests missed could not have been found with the methodology used. The implementation runs in one of two modes. In one mode, the test engineer explicitly sets a pre-state by entering the state name. In the other mode, the software always starts at the initial state, and a test case prefix must be included as part of the test case. The prefix should include inputs to reach the pre-state. All of the tests that were used in this study explicitly set the pre-state, and three of the four faults that were missed could not be found if the pre-state is explicitly set. These faults were in statements that were not executed if the prefix was explicitly set. None of the three sets of tests found these three faults. The other fault that the full predicate tests missed was not found by either of the other two sets of inputs. Of the other five faults missed by the random and the transition tests, two were the same, and the other three were different. All of the naturally occurring faults were found by all three sets of tests.

The goals of this empirical pilot study were twofold. The first goal was to see if the specification-based testing criteria could be practically applied. The second was to make a preliminary evaluation of their merits by evaluating the branch coverage and fault coverage. Both goals were satisfied; the criteria were applied and worked well. They performed better than random generation of tests. However, there are several limitations to the interpretation of the results. First, Cruise is of moderate size; longer and more complicated programs are needed. Second, the 25 faults inserted into Cruise were generated intuitively. More study should be car-

	<b>Random</b>	<b>Transition</b>	<b>Full Predicate</b>
number of test cases	54	12	54
faults found	15	15	20
faults missed	9	9	4
percent coverage	62.5%	62.5%	83.3%

Table 2: Faults Detected

ried out to reveal the types of faults that can be detected by system testing. In future studies, it is desired to have an automated test case generator to generate tests, which can be used to experiment with bigger and more complicated software. Also, other system level testing techniques should be used for comparison.

## 5 Conclusions and Future Work

This paper introduces a new technique for generating tests from formal software specifications. Formal specifications represent a significant opportunity for testing because they precisely describe the functionality of the software in a form that can be easily manipulated by automated means. This research addresses the problem of developing formalizable, measurable criteria for generating tests from specifications. Results from applying the criteria and process to a small example were presented. This case study was evaluated using Atac to measure decision coverage, and the technique was found to achieve a high level of coverage. It was also used to successfully detect a large percentage of faults. These results indicate that this technique can benefit software developers who construct formal specifications during development.

One interesting result from the decision coverage is that only the functional specifications related to the cruise control state machine itself were covered. While this was certainly the focus of the study, several decisions having to do with the input were left out. For testing of real systems, the input specifications must be considered as well, either by adapting the method presented here, or by using another testing method.

The immediate goal of this research was to develop formal criteria for generating tests from state-based specifications.

This research is ongoing in two major directions. A preliminary proof-of-concept tool has been implemented [31]. This tool currently generates full-predicate and transition-pair test cases from either SCR condition tables or UML statecharts. This tool is integrated with the Naval Research Laboratory's SCR\* Toolset [17] and Rational Software Corpora-

tion's Rational Rose tool [11]. We are currently extending this tool to remove various restrictions on the form of the specifications. We are also evaluating these criteria in terms of their usefulness to industrial applications. We are working with Rockwell-Collins to evaluate these criteria on their Flight Guidance System (FGS) [29]. We are currently carrying out an empirical study using an SCR specification for FGS (14 mode transition tables) and a Java implementation (about 115 Java classes). It is hoped that this work can demonstrate the practical usefulness of these criteria, and also help develop clear guidelines about how test managers can make cost/benefit choices about using the criteria.

## 6 Acknowledgments

We would like to thank Paul Ammann for help with the definitions of the testing criteria.

## References

- [1] P. Ammann and A. J. Offutt. Using formal methods to derive test frames in category-partition testing. In *Proceedings of the Ninth Annual Conference on Computer Assurance (COMPASS 94)*, pages 69–80, Gaithersburg MD, June 1994. IEEE Computer Society Press.
- [2] J. M. Atlee. Native model-checking of SCR requirements. In *Fourth International SCR Workshop*, November 1994.
- [3] J. 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.
- [4] M. Balcer, W. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. In *Proceedings of the Third Symposium on Software Testing, Analysis, and Verification*, pages 210–218, Key West Florida, December 1989. ACM SIGSOFT 89.

- [5] G. Bernot, M. C. Gaudel, and B. Marre. Software testing based on formal specifications: A theory and a tool. *Software Engineering Journal*, 6(6):387–405, 1991.
- [6] M. Blackburn and R. Busser. T-VEC: A tool for developing critical systems. In *Proceedings of the 1996 Annual Conference on Computer Assurance (COMPASS 96)*, pages 237–249, Gaithersburg MD, June 1996. IEEE Computer Society Press.
- [7] L. Bougé, N. Choquet, L. Fribourg, and M.-C. Gaudel. Test sets generation from algebraic specifications using logic programming. *The Journal of Systems and Software*, 6(4):343–360, November 1986.
- [8] J. Chang and D. Richardson. Structural specification-based testing with ADL. In *Proceedings of the 1996 International Symposium on Software Testing, and Analysis*, pages 62–70, San Diego, CA, January 1996. ACM Press.
- [9] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, pages 193–200, September 1994.
- [10] N. Choquet. Test data generation using a prolog with constraints. In *Proceedings of the Workshop on Software Testing*, pages 51–60, Banff Alberta, July 1986. IEEE Computer Society Press.
- [11] Rational Software Corporation. *Rational Rose 98: Using Rational Rose*. Rational Rose Corporation, Cupertino CA, 1998.
- [12] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proceedings of FME '93: Industrial-Strength Formal Methods*, pages 268–284, Odense, Denmark, 1993. Springer-Verlag Lecture Notes in Computer Science Volume 670.
- [13] R. K. Doong and P. G. Frankl. Case studies on testing object-oriented programs. In *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification*, pages 165–177, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society Press.
- [14] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.
- [15] J. Gannon, P. McMullin, and R. Hamlet. Data-abstraction implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems*, 3(3):211–223, July 1981.
- [16] I. J. Hayes. Specification directed module testing. *IEEE Transactions on Software Engineering*, SE-12(1):124–133, January 1986.
- [17] C. Heitmeyer, J. Kirby, and B. Labaw. Tools for formal specification, verification, and validation of requirements. In *Proceedings of the 1997 Annual Conference on Computer Assurance (COMPASS 97)*, pages 35–47, Gaithersburg MD, June 1997. IEEE Computer Society Press.
- [18] K. Henninger. Specifying software requirements for complex systems: New techniques and their applications. *IEEE Transactions on Software Engineering*, SE-6(1):2–12, January 1980.
- [19] Robert M. Hierons. Testing from a Z specification. *The Journal of Software Testing, Verification, and Reliability*, 7:19–33, 1997.
- [20] M. Hlady, R. Kovacevic, J. J. Li, B. R. Pekilis, D. Prairie, T. Savor, R. E. Seviora, D. Simser, and A. Vorobiev. An approach to automatic detection of software failures. In *Proceedings of the IEEE 6th International Symposium on Software Reliability Engineering (ISSRE)*, pages 314–323, Toulouse-France, October 1995.
- [21] J. R. Horgan and S. London. ATAC: A data flow coverage testing tool for C. In *Proceedings of the Symposium of Quality Software Development Tools*, pages 2–10, New Orleans LA, May 1992.
- [22] 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.
- [23] R. A. Kemmerer. Testing formal specifications to detect design errors. *IEEE Transactions on Software Engineering*, SE-11(1):32–43, January 1985.
- [24] G. Laycock. Formal specification and testing: A case study. *The Journal of Software Testing, Verification, and Reliability*, 2:7–23, 1992.
- [25] J. J. Li and R. E. Seviora. Automatic failure detection with conditional-belief supervisors. In *Proceedings of the IEEE 7th International Symposium on Software Reliability Engineering (ISSRE 96)*, pages 4–13, White Plains, NY, October 1996.

- [26] S. Liu, A. J. Offutt, C. Ho-Stuart, Y. Sun, and Mitsuru Ohba. SOFL: A formal engineering methodology for industrial applications. *IEEE Transactions on Software Engineering*, 24(1):337–344, January 1998. Special Issue on Formal Methods.
- [27] G. Luo, G. v. Bochmann, and A. Petrenko. Test selection based on communicating nondeterministic finite-state machines using a generalized Wp-method. *IEEE Transactions on Software Engineering*, 20(2):149–162, February 1994.
- [28] Luqi, H. Yang, and X. Zhang. Constructing an automated testing oracle: An effort to produce reliable software. In *Proceedings of IEEE Conference on Computer Software and Applications (COMPSAC)*, 1994.
- [29] S. P. Miller and K. F. Hoech. Specifying the mode logic of a flight guidance system in core. Release - 1 - 1, Collins Commercial Avionics, Rockwell Collins, Inc., Cedar Rapids, IA, 1997.
- [30] A. J. Offutt and S. Liu. Generating test data from SOFL specifications. *The Journal of Systems and Software*, 1999. To Appear.
- [31] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In *Proceedings of the Second IEEE International Conference on the Unified Modeling Language (UML99)*, Fort Collins, CO, October 1999. IEEE Computer Society Press.
- [32] T. J. Ostrand, R. Sigal, and E. J. Weyuker. Design for a tool to manage specification-based testing. In *Proceedings of the Workshop on Software Testing*, pages 41–50, Banff Alberta, July 1986. IEEE Computer Society Press.
- [33] RTCA Committee SC-167. Software considerations in airborne systems and equipment certification, Seventh draft to Do-178A/ED-12A, July 1992.
- [34] I. Spence and C. Meudec. Generation of software tests from specifications.
- [35] P. Stocks and D. Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, November 1996.
- [36] W. T. Tsai, D. Volovik, and T. F. Keefe. Automated test case generation for programs specified by relational algebra queries. *IEEE Transactions on Software Engineering*, 16(3), March 1990.
- [37] E. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, May 1994.