
Generating Test Data From State-based Specifications¹

A. Jefferson Offutt¹, Shaoying Liu², and Aynur Abdurazik¹

¹*ISE Department, Software Engineering Research Lab, George Mason University, Fairfax, VA 22030, USA*

²*Faculty of Information Sciences, Hosei University, 3-7-2 Kajino-cho Koganei-shi, Tokyo 184-8584 Japan*

**** DRAFT *** Submitted to JSTVR ****

ABSTRACT

Although the majority of software testing in industry is conducted at the system level, most formal research has focused on the unit level. As a result, most system level testing techniques are only described informally. This paper presents formal testing criteria for system level testing that are based on formal specifications of the software. Software testing can only be formalized and quantified when a solid basis for test generation can be defined. Formal specifications represent a significant opportunity for testing because they precisely describe what functions the software is supposed to provide in a form that can be easily manipulated.

This paper presents general criteria for generating test inputs from state-based specifications. The criteria include techniques for generating tests at several levels of abstraction for specifications. 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.

KEY WORDS: Formal Methods, Specification-based Testing, Software Testing.

¹This work is supported in part by Rockwell Collins, Inc, in part by the U.S. National Science Foundation under grant CCR-98-04111, and in part by the Ministry of Education of Japan under Joint Research Grant-in-Aid for International Scientific Research FM-ISEE (08044167).

1 Introduction

There is an increasing need for effective testing of software for 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 test data from formal specifications; this paper presents results for generating test data from one kind of formal specifications. Formal specifications 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 by automated means.

This paper presents a model for developing test inputs from state-based specifications, and formal criteria for test data selection. Techniques for developing test inputs that are derived from SOFL specifications [LOHS⁺98] are presented elsewhere [OL99]. The test data 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. An abstract view of part of a typical test process that might be used for code-based test generation is summarized by the diagram in Figure 1 (A). 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*, an abstract view of which is shown in Figure 1 (B). Here the specifications are used to produce test cases, as well as to produce the program. In this scenario, the specifications are more likely to be formalized, so the arc from S to P is

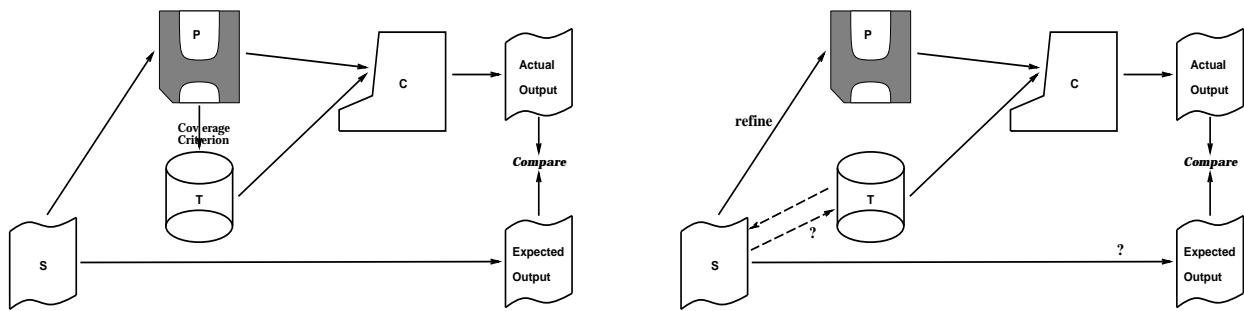


Figure 1: (A) Code-based Test Generation

(B) Specification-based Test Generation

labeled “refine”, to indicate a refinement process, which can be used to create code from formal specifications. One significance of producing 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, the test engineer will often find inconsistencies and ambiguities in the specifications when the tests are generated, allowing the specifications to be improved before the program is written (hence the feedback arc from T back to S). The arcs from S to T and from S to $ExpectedOutput$ are labeled with a “?”, because these are currently areas of active research. 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 [LS96, HKL+95, LYZ94].

Specification-based test data 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 test data 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 case generation, to check the output of software on test inputs [LYZ94], and as a basis for formalizing *test specifications* (as opposed to functional specifications) [SC93a, SC96, SC93b]. This paper is primarily concerned with the first use, that of generating test cases from specifications, which is referred to as specification-based testing. An immediate

goal is to develop *mechanical* procedures to derive test cases from formal specifications; long term goals include automated tool support to transform formal functional specifications into effective test cases.

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 testing 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 quality 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 presents new criteria for generating tests from state-based specifications. Applications of these criteria to three different specification languages are discussed, examples are presented using Software Cost Reduction specifications (SCR) [Hen80, AG93] and CoRE [FBWK92], and empirical results from a case study are shown. This paper also includes a review of the small but growing body of work on using formal specifications as a basis for producing test cases.

2 Specification-based Testing Criteria

An important problem in software testing is deciding when to stop. 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 [FW88].

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. These criteria are defined on state-based specifications. 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 [Hen80, AG93] 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.

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 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.

There are 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. To apply these, a state-based requirement/specification is viewed as a directed graph, called the *specification graph*. Each node represents a state (or mode) in the requirement/specification, and edges represent possible transitions among states.

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 test cases 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.

2.1 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 .

2.2 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 [SC-92] 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.

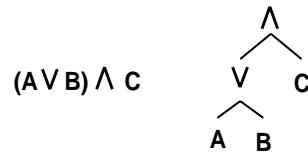
Full predicate coverage: For each predicate P on each transition, **T** must include tests that cause each clause c in P to result in a pair of outcomes where the value of P is directly correlated with the value of c .

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**.

2.2.1 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. 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 2 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**):

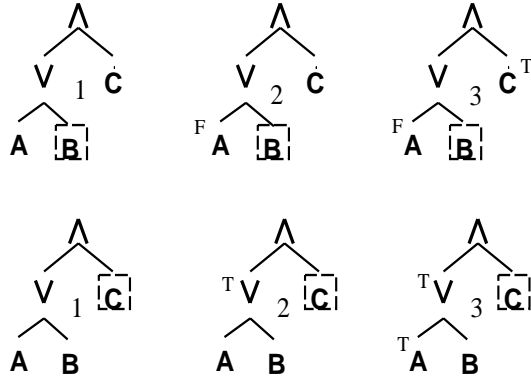


Figure 2: Constructing Test Case Requirements From an Expression Parse Tree

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

2.2.2 Handling triggering events

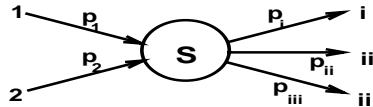
As defined in Section 5.3, a triggering event is a change in a value for a variable, expression, or expressions that causes the software to transition from one state to another. In SCR and CoRE, triggering event variables have a different format from other variables in transition predicates. A triggering 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.

2.3 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, this criterion 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 $(P_2:P_{iii})$.

2.4 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. This use of the term is similar to that of “use cases”. 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.

2.5 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. Transition coverage is similar to branch

coverage. Full predicate coverage relies on definitions from DO178B [SC-92], and the definition is similar to that of modified condition/decision coverage (MC/DC) [CM94], 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 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 Automatically Deriving Test Cases

This section describes a tool, `SPECTEST`, which generates test cases according to the criteria defined in Section 2. `SPECTEST` is defined in terms of its process, and the processes for all four criteria are presented together, as there is a fair amount of overlap. The process overview is shown in Figure 3, and the steps that `SPECTEST` follows are described below. `SPECTEST` currently works for SCR specifications created by using the `SCRTool` developed at the Naval Research Laboratory [HKL97], and the UML Rational Software Corporation’s Rational Rose tool [Cor98].

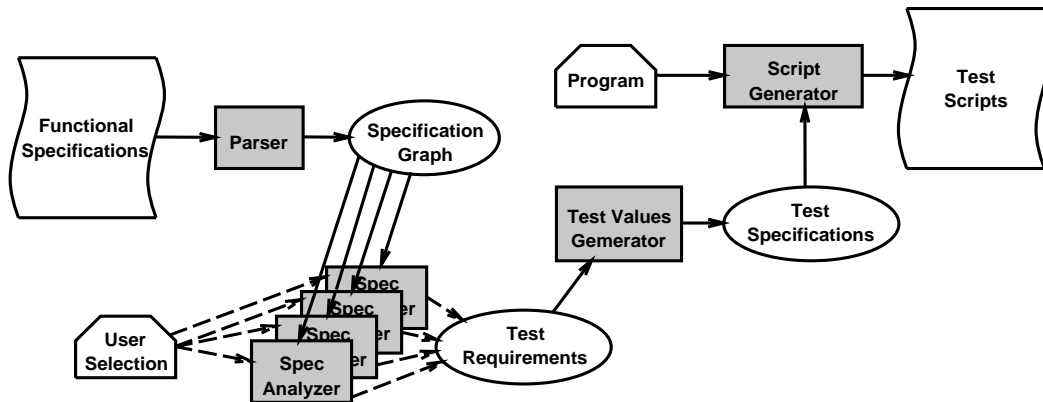


Figure 3: General Process for Generating Test Cases

1. **Parse functional specifications.** **Transition conditions** are predicates that define under what conditions each transition will be taken. With some specification languages (e.g., SCR and CoRE), the transition conditions are encoded directly into the specifications, otherwise they have to be derived. `SPECTEST` reads transition conditions directly from SCR tables and from UML tables. For languages that do not directly encode transition conditions, the transition conditions could be derived by hand or generated by a separate pre-processing tool.

2. **Develop specification graph.** The specification graph can be directly derived from the transition conditions, and edges annotated with the conditions derived in step 1.

At this point, the process separates for the four testing criteria. The user selects which criterion to apply.

3. **Develop transition coverage test requirements.**

- (a) **Derive transition predicates.** The conditions from step 1 are listed one at a time to form test requirements.

4. **Develop full predicate test requirements.**

- (a) **Construct truth tables for all predicates in the specification graph.** If all the logical connectors are the same (all ANDs or all ORs), it is a simple matter to modify the values for the clauses in the predicates directly. If ANDs and ORs are mixed freely, however, it is less error-prone to construct the expression tree. SPECTEST creates explicit parse trees. Some specification languages differentiate between trigger events and preconditions; in this case, the trigger events must be marked specially so that the trigger event values appear after the precondition inputs.

5. **Develop transition-pair test requirements.**

- (a) **Identify all pairs of transitions.** Transition-pair tests are ordered pairs of condition values, each representing an input to the state and an output from the state. These are formed by enumerating all the input transitions (M), all the output transitions (N), then creating $M * N$ pairs of transitions.
- (b) **Construct predicate pairs.** These pairs of transitions are then replaced by the predicates from the specification graph.

6. **Develop complete sequence test requirements.**

- (a) **Identify complete lists of states.** The complete sequence tests are created by the tester. This is done by choosing sequences of states from the specification graph to enter.
- (b) **Construct sequence of predicates.** The sequences of states are transformed into sequences of conditions that will cause those states to be entered.

At this point, test requirements for the four criteria will be in a uniform format: truth assignments for predicates. These truth assignments form the test requirements for the testing.

7. **Test Values Generator.** For each unique test requirement, generate test specifications that consist of prefix values, test case values, verify conditions, exit conditions, and expected outputs. Note that there may be a fair amount of overlap among the test requirements, thus the “unique” restriction. Before test specifications are generated, duplicate test requirements are removed. Generating the actual values involve solving some algebraic equations. For example, if a condition is $A > B$, values for A and B must be chosen to give the predicate the appropriate value. It is also at this point that some “invalid” tests might be discovered. For example, it may be impossible or meaningless to pair all incoming and outgoing transitions for each state. Such test specifications are discarded.
8. **Construct test scripts.** Each test specification is used to construct one test script. The actual scripts must reflect the input syntax of the program. (Note that this is the only step that requires any knowledge of the implementation, all preceding steps depend solely on the functional specifications.)

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.

SPECTEST currently has the restriction that it processes only one mode class for SCR specification at a time, and one statechart class for UML specifications [OA99]. The tool is currently being expanded to handle more than one mode class and state chart. It is also being expanded to handle other UML diagrams, most currently including collaboration diagrams.

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. The methodology and empirical subjects are described first, then the processes used to generate tests for each criterion are described in detail. Then the implementation used, and the faults that were generated are described, and finally results and analysis are given.

4.1 Methodology

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, representative faults were seeded, and test cases were generated by hand.

Cruise control is a common example in the literature [Atl94, Jin96], and specifications are readily available. The specifications used are given in Table 1. 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*.

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	-	-	-	-	Override
	t	t	f	@T	-	-	-	
	t	t	f	-	-	@T	-	
Override	@F	-	-	-	-	-	-	Off
	t	@F	-	-	-	-	-	Inactive
	t	t	-	f	@T	-	-	Cruise
	t	t	-	f	-	-	@T	

Table 1: SCR Specifications for the Cruise Control System

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[¬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).

Table 2 shows the transitions of the specification with the trigger events expanded in predicate form, numbered P_1 through P_{12} . Figure 4 shows the specification graph, with the edges labeled with the predicate numbers.

P_1	OFF	$\neg Ignited \wedge Ignited'$	INACTIVE
P_2	INACTIVE	$Ignited \wedge \neg Ignited'$	OFF
P_3	INACTIVE	$\neg Activate \wedge Ignited \wedge Running \wedge \neg Brake \wedge Activate'$	CRUISE
P_4	CRUISE	$Ignited \wedge \neg Ignited'$	OFF
P_5	CRUISE	$Running \wedge Ignited \wedge \neg Running'$	INACTIVE
P_6	CRUISE	$\neg Toofast \wedge Ignited \wedge Toofast'$	INACTIVE
P_7	CRUISE	$\neg Brake \wedge Ignited \wedge Running \wedge \neg Toofast \wedge Brake'$	OVERRIDE
P_8	CRUISE	$\neg Deactivate \wedge Ignited \wedge Running \wedge \neg Toofast \wedge Deactivate'$	OVERRIDE
P_9	OVERRIDE	$Ignited \wedge \neg Ignited'$	OFF
P_{10}	OVERRIDE	$Running \wedge Ignited \wedge \neg Running'$	INACTIVE
P_{11}	OVERRIDE	$\neg Activate \wedge Ignited \wedge Running \wedge \neg Brake \wedge Activate'$	CRUISE
P_{12}	OVERRIDE	$\neg Resume \wedge Ignited \wedge Running \wedge \neg Brake \wedge Resume'$	CRUISE

Table 2: Expanded Cruise Control Specification Predicates

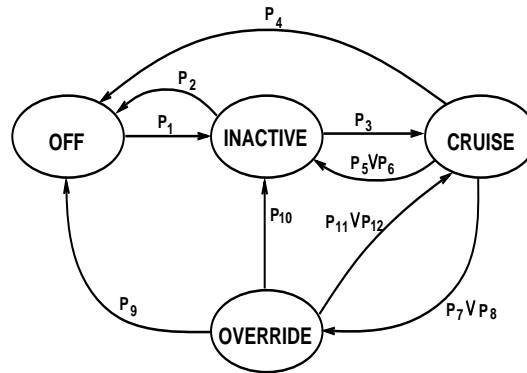


Figure 4: Specification Graph for Cruise Control

4.2 Test Generation

This case study was also used to validate the test data generation process, so tests were generated by hand. 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. The number of faults detected was recorded and used in the analysis. The rest of this subsection discusses in detail how tests were generated. This serves both to elaborate on the empirical methodology, as well as to illustrate the criteria defined previously. Because the transition coverage criterion is subsumed by full predicate coverage, it is not described separately. Transition coverage test cases can be taken from the “valid” specifications in full predicate, which are listed first for each transition.

4.2.1 Full predicate coverage criterion

There are nine transitions in the cruise control specifications, and twelve disjunctive predicates. For convenience, the technique is applied by considering each predicate specification separately. As described in Section 2.2, both the before-values and after-values of the triggering event should be separately tested. For SCR, this is handled by treating @ as an operator and expanding it algebraically. If X represents a before-value and X' an after-value, the relevant expansions are:

- $@T(X) \equiv NOTX \wedge X'$
- $@T(X \wedge Y) \equiv \neg(X \wedge Y) \wedge (X' \wedge Y') \equiv (\neg X \vee \neg Y) \wedge X' \wedge Y'$
- $@T(X \vee Y) \equiv \neg(X \vee Y) \wedge (X' \vee Y') \equiv \neg X \wedge \neg Y \wedge (X' \vee Y')$

There are 54 separate test case requirements for the full predicate coverage level (they are listed in Appendix A). The third transition, P_3 , is used to illustrate the test case requirement derivation. The variable values are taken from the predicates, and are shown as T, F, t, f, and -. A T or F means the clause is triggering, and the table contains a before-value and after-value. The values for the test case are the new value for the triggering clause (T or F), and the t and f values from the WHEN conditions. The expected output for the test specification is derived from the triggering event, the post-state, and any terms or variables that are defined as a result of the transition. P_3 has four clauses:

$$@TActivate \wedge Ignited \wedge Running \wedge \neg Brake$$

and its expanded version is:

$$\neg Activate \wedge Ignited \wedge Running \wedge \neg Brake \wedge Activate'$$

Its six test case requirements are:

Pre State	<i>Activate</i>	<i>Ignited</i>	<i>Running</i>	<i>Brake</i>	<i>Activate'</i>	Post State
1. INACTIVE	F	t	t	f	T	CRUISE
2. INACTIVE	F	f	t	f	T	INACTIVE
3. INACTIVE	F	t	f	f	T	INACTIVE
4. INACTIVE	F	t	t	t	T	INACTIVE
5. INACTIVE	T	t	t	f	T	INACTIVE
6. INACTIVE	F	t	t	f	F	INACTIVE

The first row is the predicate as it appears in the specification; every clause is **True**. This corresponds to a valid test input (and is also the transition coverage test case for this transition). The subsequent rows make each clause **False** in turn, corresponding to invalid inputs. Because there are no OR operators, the full predicate coverage criterion is satisfied by holding all other clauses **True**. The post-states are the expected values. Five of them represent invalid transitions, and it is assumed that the software will remain in the same state.

Test specifications

The actual test specifications and test scripts are mechanically derived from the test requirements. The predicate P3 is chosen as an illustrative example. P3 has six full predicate level tests. For the first test case for P3, the test case must reach the INACTIVE state; this forms the **Prefix**. The **Test case values** set the before-value for the triggering event, and the WHEN condition variables of *Inactive*, *Running*, and *Brake*, and then sets *Activate* to be **True** as the triggering event. The **Verify** and **Exit** parts of the specifications are not shown, as they depend on the software. The software can safely be assumed to automatically print the current state, and to not require an exit.

1. Test specification P3-1:

Prefix:	<i>Ignited</i>	= True	– Reach INACTIVE state
Test case value:	<i>Activate</i>	= False	– Trigger before-value
	<i>Running</i>	= True	– Condition variable
	<i>Brake</i>	= False	– Condition variable
	<i>Activate</i>	= True	– Triggering event
Expected outputs:	CRUISE		

2. Test specification P3-2:

Prefix: *Ignited* = **True** – Reach INACTIVE state
Test case value: *Activate* = **True** – Trigger before-value
Running = **True** – Condition variable
Brake = **False** – Condition variable
Activate = **True** – Triggering event
Expected outputs: INACTIVE

3. Test specification P3-3:

Prefix: *Ignited* = **True** – Reach INACTIVE state
Test case value: *Activate* = **False** – Trigger before-value
Ignited = **False** – Condition variable
Running = **True** – Condition variable
Brake = **False** – Condition variable
Activate = **True** – Triggering event
Expected outputs: INACTIVE

4. Test specification P3-4:

Prefix: *Ignited* = **True** – Reach INACTIVE state
Test case value: *Activate* = **False** – Trigger before-value
Running = **False** – Condition variable
Brake = **False** – Condition variable
Activate = **True** – Triggering event
Expected outputs: INACTIVE

5. Test specification P3-5:

Prefix: *Ignited* = **True** – Reach INACTIVE state
Test case value: *Activate* = **False** – Trigger before-value
Running = **True** – Condition variable
Brake = **True** – Condition variable
Activate = **True** – Triggering event
Expected outputs: INACTIVE

6. Test specification P3-6:

Prefix: *Ignited* = **True** – Reach INACTIVE state
Test case value: *Activate* = **False** – Trigger before-value
Running = **True** – Condition variable
Brake = **False** – Condition variable
Activate = **False** – Triggering event
Expected outputs: INACTIVE

There are several interesting points to note about these test specifications. First, it should be clear that there is some redundancy; some of the condition variables do not need to be explicitly set, as they will already have the appropriate values. Algorithms necessary to decide what values do and do not need to be are found in our technical report [Off99]; these algorithms are implemented in `SPECTEST`.

Another interesting point is the derivation of the prefix part of the test specification. Reaching the pre-state is essentially a reachability problem. Given a control flow graph of a program, it is an undecidable problem to find a test case that reaches a particular statement. state-based systems are finite and deterministic, so this problem is solvable for specification graphs derived from state-based systems. An algorithm for doing this is also in our technical report [Off99], and prefixes are generated automatically by `SPECTEST`.

Test scripts are simple rewrites of test specifications with modifications made for the input requirements of the program being tested. The test script for the first test specification above is:

```
Ignited = True
Activate = False
Running = True
Brake = False
Activate = True
```

4.2.2 Transition-pair coverage criterion

At the transition-pair level, each state is considered separately. Each input transition into the state is matched with each transition out of the state, and the combination is used to create test requirements, which are ordered pairs of predicates. The ordered pairs are turned into ordered pairs of inputs to form test specifications.

Following are the test requirements for the four states.

OFF	CRUISE
1. P2 : P1 2. P4 : P1 3. P9 : P1	1. P3 : P4 2. P3 : (P5 OR P6) 3. P3 : (P7 OR P8)
INACTIVE	4. (P11 OR P12) : P4 5. (P11 OR P12) : (P5 OR P6) 6. (P11 OR P12) : (P7 OR P8)
1. P1 : P2 2. P1 : P3 3. P10 : P2 4. P10 : P3 5. (P5 OR P6) : P2 6. (P5 OR P6) : P3	OVERRIDE
	1. (P7 OR P8) : P9 2. (P7 OR P8) : P10 3. (P7 OR P8) : (P11 OR P12)

These ordered pairs are transformed into predicates from Table 2. The “**OR**” entries result from the transitions that have two conditions; either condition could be satisfied to take that transition. The complete set of resulting predicates are shown in Appendix B; they result in 36 additional test cases.

Test specifications

The actual test specifications and test scripts are mechanically derived from the above test requirements, and are too numerous to list. The requirements for the OFF state are chosen as an illustrative example. OFF has three transition-pair coverage level tests. For the first test case for OFF, the test case must reach the INACTIVE state; this forms the **Prefix**. Then the test case must pass through transitions *P1* and *P2*.

1. Test specification OFF-1:

Prefix: *Ignited* = **True** – Reach INACTIVE state
 Test case values: *Ignited* = **False** – P2 Triggering event
 Ignited = **True** – P1 Triggering event
 Expected outputs: INACTIVE

2. Test specification OFF-2:

Prefix: *Ignited* = **True** – Reach INACTIVE state
 Ignited = **True** – P3 Condition variable
 Running = **True** – P3 Condition variable
 Brake = **False** – P3 Condition variable
 Activate = **True** – Reach CRUISE state
 Test case values: *Ignited* = **False** – P4 Triggering event
 Ignited = **True** – P1 Triggering event
 Expected outputs: INACTIVE

3. Test specification OFF-3:

Prefix: *Ignited* = **True** – Reach INACTIVE state
 Ignited = **True** – P3 Condition variable
 Running = **True** – P3 Condition variable
 Brake = **False** – P3 Condition variable
 Activate = **True** – Reach CRUISE state
 Ignited = **True** – P7 Condition variable
 Running = **True** – P7 Condition variable
 Toofast = **False** – P7 Condition variable
 Brake = **True** – Reach OVERRIDE state
 Test case values: *Ignited* = **False** – P9 Triggering event
 Ignited = **True** – P1 Triggering event
 Expected outputs: INACTIVE

4.2.3 Complete sequence criteria

At the complete sequence level, test engineers must use their experience and judgment to develop sequences of states that should be tested. To do this well requires experience with testing, experience with programming, and knowledge of the domain. These tests are omitted in this case study.

4.3 Implementation and Faults

A model of the cruise control problem was implemented in about 400 lines of C. Cruise has seven functions, 184 blocks, and 174 decisions. The program accepts pairs of variable:values, where a value can be 't', 'f', 'T', or 'F'. Upper case inputs signify a triggering event. For convenience, the program was implemented so that the pre-state could be either set with a test case `Prefix`, or explicitly by entering the name of a state.

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.

4.4 Results and Analysis

As a way to measure the quality of these tests, block and decision coverage was computed using the full predicate test cases. The coverage was measured using Atac [HL92]. Of the 174 decisions, 5 are infeasible, leaving 169. The results are shown in Figure 5. 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 test data are shown in Table 3.

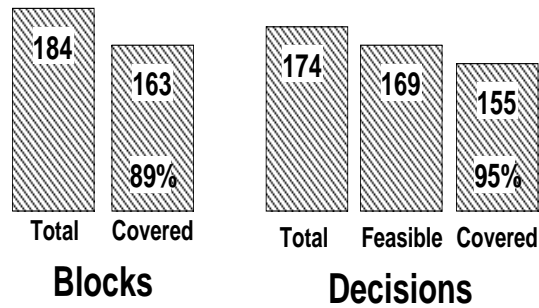


Figure 5: Branch and Decision Coverage Results

	Random	Transition	Full Predicate
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 3: Faults Detected

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 test cases. 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 carried out to reveal the types of faults that can be detected by system testing.

5 Related Work

The current research literature reports on specific tools for specific formal specification languages [BGM91, BCFG86, GMH81, Jal92, OSW86, TVK90, WGS94], manual methods for deriving tests from specifications [AA92, AO94, Ber91, DF93, Hay86], case studies on using specifications to check the output of the software on specifications [DF91, Kem85, Lay92, SC93a], and formalizations of test specifications [SC96, SC93b, BHO89, Cho86]. The term *specification-based testing* is used in the narrow sense of using specifications as a basis for deciding what tests to run on software. This section reviews some of these techniques, dividing them into approaches that use model-based, algebraic, and state-based specifications. It should be noted that this review is not complete; only a sampling of the most relevant research can fit into this section.

5.1 Model-based Approaches

Model-based specification languages, such as Z and VDM, attempt to derive formal specifications of the software based on mathematical models. Spence and Meudec [SM] and Dick and Faivre [DF93] suggested using specifications to produce predicates, and then using predicate satisfaction techniques to generate test data. Given a set of predicates that reflect preconditions, invariants, and postconditions, test cases are generated to satisfy individual clauses. Their work was for VDM specifications, and primarily focused on state-based specifications, using finite state automata representations. Dick and Faivre discussed straightforward translations of the specifications into disjunctive normal form predicates, and presented solutions to the problem of predicate satisfaction by using prolog theorem proving techniques. Their work does not include testing with invalid inputs, specifically testing for faults, or comprehensive criteria for test selection and measurement.

Stocks and Carrington [SC93b, SC93a] and Amla, Ammann, and Offutt [AA92, AO94] proposed using a form of *domain partitioning* to generate test cases. Given a description of an input domain, the idea is to use specifications to partition the input domain into subsets. The Amla, Ammann, and Offutt approach is based on a modification to the category-partition method for test generation [BHO89, OSW86]. Hierons [Hie97] presents algorithms that rewrite Z specifications into a form that can be used to partition the input domain. From this, states of a finite state automaton are derived, which is then used to control the test process.

Hayes [Hay86] has suggested a dynamic scheme that uses *run-time verification* of the program. The idea is to add code to the program to check predicates from the specifications, such as type

invariants, preconditions, and input-output pairs.

Chang and Richardson [CR96] presented techniques to derive test conditions from ADL specification, a predicate logic-based language that is used to describe the relationships between inputs and outputs of a program unit. The idea is to use test selection strategies to partition both input and output domains.

5.2 Algebraic Approaches

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 [GMH81] used a *script derivation* approach. They treated the axioms as a language description and generated strings on that language to serve as test cases. Doong and Frankl [DF91] used a similar approach to test object-oriented software.

Bernot [Ber91] proposed a similar scheme, with more formalization of the process and the test cases. Bougé et al. [BCFG86] suggested a logic programming approach to generating test cases from algebraic specifications. Tsai, Volovik, and Keefe [TVK90] used a similar approach, but started with relational algebra queries.

5.3 State-based Approaches

Specification-mutation testing is defined with respect to a model checking specification [ABM98, AB99]. Mutation analysis of specifications yields mutants from which a model checker generates counterexamples that can be used as test cases. 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 of avoiding brute force exploration of the state space.

Blackburn [BB96] used state-based functional specifications of the software, expressed in the language **T-Vec**, to derive disjunctive normal form constraints, similarly to Dick and Faivre's method. These constraints are then solved to generate test cases, using special-purpose heuristic algorithms. There is a strong similarity to Blackburn's algorithms and the algorithms used by Offutt's test data generator [DGK⁺88, DO91]; the key difference being that Blackburn's is specification-based, whereas Offutt's constraints are code-based.

Weyuker, Goradia, and Singh [WGS94] present a method to generate test data from boolean logic specifications of software. They applied their techniques to the FAA's Traffic Collision and

Avoidance System (TCAS), and used a few mutation-style faults to measure the quality of the test cases. Their method is very similar to the full predicate criterion we describe, but are restricted to boolean variables.

5.4 Summary

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.

6 Conclusions

This paper introduces a new technique for generating test data 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 test cases from specifications. Criteria tests from requirements/specifications and a derivation process for generating the test cases were presented. 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. Short term goals are to develop *mechanical* procedures and an automatic test data generation tool. Longer term goals include applying these criteria to industrial software and to expand SPECTEST to handle other UML diagrams and other specification languages. The advantage of this automation is that it allows large systems to be tested.

References

- [AA92] N. Amla and P. Ammann. Using Z specifications in category partition testing. In *Proceedings of the Seventh Annual Conference on Computer Assurance (COMPASS*

- 92), Gaithersburg MD, June 1992. IEEE Computer Society Press.
- [AB99] 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.
- [ABM98] 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.
- [AG93] 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.
- [AO94] 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.
- [At194] J. M. Atlee. Native model-checking of SCR requirements. In *Fourth International SCR Workshop*, November 1994.
- [BB96] 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.
- [BCFG86] 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.
- [Ber91] G. Bernot. Testing against formal specifications: A theoretical view. Technical report LIENS-91-1, LIENS, Département de Mathématiques et d’Informatique, January 1991.
- [BGM91] 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.
- [BHO89] 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.
- [Cho86] 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.
- [CM94] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, pages 193–200, September 1994.
- [Cor98] Rational Software Corporation. *Rational Rose 98: Using Rational Rose*. Rational Rose Corporation, Cupertino CA, 1998.
- [CR96] 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.

- [DF91] 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.
- [DF93] 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.
- [DGK⁺88] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An extended overview of the Mothra software testing environment. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 142–151, Banff Alberta, July 1988. IEEE Computer Society Press.
- [DO91] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.
- [FBWK92] S. Faulk, J. Brackett, P. Ward, and J. Kirby. The CoRE method for real-time requirements. *IEEE Software*, pages 22–33, September 1992.
- [FW88] 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.
- [GMH81] 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.
- [Hay86] I. J. Hayes. Specification directed module testing. *IEEE Transactions on Software Engineering*, SE-12(1):124–133, January 1986.
- [Hen80] 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.
- [Hie97] Robert M. Hierons. Testing from a Z specification. *The Journal of Software Testing, Verification, and Reliability*, 7:19–33, 1997.
- [HKL⁺95] M. Hlady, R. Kovacevic, J. J. Li, B. R. Pekilis, D. Prairie, T. Savor, R. E. Seviara, 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.
- [HKL97] 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.
- [HL92] 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.
- [Jal92] P. Jalote. Specification and testing of abstract data types. *Computer Language*, 17(1):75–82, 1992.

- [Jin96] 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.
- [Kem85] R. A. Kemmerer. Testing formal specifications to detect design errors. *IEEE Transactions on Software Engineering*, SE-11(1):32–43, January 1985.
- [Lay92] G. Laycock. Formal specification and testing: A case study. *The Journal of Software Testing, Verification, and Reliability*, 2:7–23, 1992.
- [LOHS⁺98] 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.
- [LS96] 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.
- [LYZ94] 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.
- [OA99] 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.
- [Off99] A. J. Offutt. Generating test data from requirements/specifications: Phase II final report. Technical report ISE-TR-99-01, Department of Information and Software Engineering, George Mason University, Fairfax VA, January 1999. <http://www.ise.gmu.edu/techrep>.
- [OL99] Jeff Offutt and Shaoying Liu. Generating test data from SOFL specifications. *The Journal of Systems and Software*, 49(1):49–62, December 1999.
- [OSW86] 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.
- [SC-92] RTCA Committee SC-167. Software considerations in airborne systems and equipment certification, Seventh draft to Do-178A/ED-12A, July 1992.
- [SC93a] P. Stocks and D. Carrington. The ISDM case study: A dependency management system (specification-based testing). Technical report, The University of Queensland, Department of Computer Science, 1993.
- [SC93b] P. Stocks and D. Carrington. Test Templates: A Specification-Based Testing Framework. In *Proceedings of the 15th International Conference on Software Engineering*, pages 405–414, Baltimore, MD, May 1993.
- [SC96] P. Stocks and D. Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, November 1996.
- [SM] I. Spence and C. Meudec. Generation of software tests from specifications.

- [TVK90] 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.
- [WGS94] 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.

A Appendix: Full Predicate Test Requirements for Cruise

The test case requirements for the full predicate coverage level show the environmental variables as I (Ignited) R (Running), T (Toofast), B (Brake), A (Activate), D (Deactivate), and S (Resume). The variable values are taken from the predicates, and are shown as T, F, t, f, and -. A T or F means the clause is triggering, and the table contains a before-and after-value. The values for the test case are the new value for the triggering clause (T or F), and the t and f values from the WHEN conditions. The expected output for the test specification is derived from the triggering event, the post-state, and any terms or variables that are defined as a result of the transition. There are 54 test cases for the 12 predicates.

	Pre State	Variable Values	Triggering Event	Post State
		I R T B A D S		
P1	OFF	F - - - - -	<i>Ignited'</i> = True	INACTIVE
	OFF	T - - - - -	<i>Ignited'</i> = True	OFF
	OFF	F - - - - -	<i>Ignited'</i> = False	OFF
P2	INACTIVE	T - - - - -	<i>Ignited'</i> = False	OFF
	INACTIVE	F - - - - -	<i>Ignited'</i> = False	INACTIVE
	INACTIVE	T - - - - -	<i>Ignited'</i> = True	INACTIVE
P3	INACTIVE	t t - f F - -	<i>Activate'</i> = True	CRUISE
	INACTIVE	f t - f F - -	<i>Activate'</i> = True	INACTIVE
	INACTIVE	t f - f F - -	<i>Activate'</i> = True	INACTIVE
	INACTIVE	t t - t F - -	<i>Activate'</i> = True	INACTIVE
	INACTIVE	t t - f T - -	<i>Activate'</i> = True	INACTIVE
	INACTIVE	t t - f F - -	<i>Activate'</i> = False	INACTIVE
P4	CRUISE	T - - - - -	<i>Ignited'</i> = False	OFF
	CRUISE	F - - - - -	<i>Ignited'</i> = False	CRUISE
	CRUISE	T - - - - -	<i>Ignited'</i> = True	CRUISE
P5	CRUISE	t T - - - - -	<i>Running'</i> = False	INACTIVE
	CRUISE	f T - - - - -	<i>Running'</i> = False	CRUISE
	CRUISE	t F - - - - -	<i>Running'</i> = False	CRUISE
	CRUISE	t T - - - - -	<i>Running'</i> = True	CRUISE
P6	CRUISE	t - F - - - -	<i>Toofast'</i> = True	INACTIVE
	CRUISE	f - F - - - -	<i>Toofast'</i> = True	CRUISE
	CRUISE	t - T - - - -	<i>Toofast'</i> = True	CRUISE
	CRUISE	t - F - - - -	<i>Toofast'</i> = False	CRUISE
P7	CRUISE	t t f F - - -	<i>Brake'</i> = True	OVERRIDE
	CRUISE	f t f F - - -	<i>Brake'</i> = True	CRUISE
	CRUISE	t f f F - - -	<i>Brake'</i> = True	CRUISE
	CRUISE	t t t F - - -	<i>Brake'</i> = True	CRUISE
	CRUISE	t t f T - - -	<i>Brake'</i> = True	CRUISE
	CRUISE	t t f F - - -	<i>Brake'</i> = False	CRUISE

P8	CRUISE	t	t	f	-	-	F	-	$Deactivate' = True$	OVERRIDE
	CRUISE	f	t	f	-	-	F	-	$Deactivate' = True$	CRUISE
	CRUISE	t	f	f	-	-	F	-	$Deactivate' = True$	CRUISE
	CRUISE	t	t	t	-	-	F	-	$Deactivate' = True$	CRUISE
	CRUISE	t	t	f	-	-	T	-	$Deactivate' = True$	CRUISE
	CRUISE	t	t	f	-	-	F	-	$Deactivate' = False$	CRUISE
P9	OVERRIDE	T	-	-	-	-	-	-	$Ignited' = False$	OFF
	OVERRIDE	F	-	-	-	-	-	-	$Ignited' = False$	OVERRIDE
	OVERRIDE	T	-	-	-	-	-	-	$Ignited' = True$	OVERRIDE
P10	OVERRIDE	t	T	-	-	-	-	-	$Running' = False$	INACTIVE
	OVERRIDE	f	T	-	-	-	-	-	$Running' = False$	OVERRIDE
	OVERRIDE	t	F	-	-	-	-	-	$Running' = False$	OVERRIDE
	OVERRIDE	t	T	-	-	-	-	-	$Running' = True$	OVERRIDE
P11	OVERRIDE	t	t	-	f	F	-	-	$Activate' = True$	CRUISE
	OVERRIDE	f	t	-	f	F	-	-	$Activate' = True$	OVERRIDE
	OVERRIDE	t	f	-	f	F	-	-	$Activate' = True$	OVERRIDE
	OVERRIDE	t	t	-	t	F	-	-	$Activate' = True$	OVERRIDE
	OVERRIDE	t	t	-	f	T	-	-	$Activate' = True$	OVERRIDE
	OVERRIDE	t	t	-	f	F	-	-	$Activate' = False$	OVERRIDE
P12	OVERRIDE	t	t	-	f	-	-	F	$Resume' = True$	CRUISE
	OVERRIDE	f	t	-	f	-	-	F	$Resume' = True$	OVERRIDE
	OVERRIDE	t	f	-	f	-	-	F	$Resume' = True$	OVERRIDE
	OVERRIDE	t	t	-	t	-	-	F	$Resume' = True$	OVERRIDE
	OVERRIDE	t	t	-	f	-	-	T	$Resume' = True$	OVERRIDE
	OVERRIDE	t	t	-	f	-	-	F	$Resume' = False$	OVERRIDE

B Appendix: Transition-pair Test Requirements for Cruise

Rather than list before-values and after-values for the triggering events in this table, only the after-values are shown; the before-values are assumed to be the inverse.

		I	R	T	B	A	D	S		
OFF:	1. INACTIVE	F	-	-	-	-	-	-	OFF	
	OFF	T	-	-	-	-	-	-	INACTIVE	
	2. CRUISE	F	-	-	-	-	-	-	OFF	
	OFF	T	-	-	-	-	-	-	INACTIVE	
	3. OVERRIDE	F	-	-	-	-	-	-	OFF	
	OFF	T	-	-	-	-	-	-	INACTIVE	
INACTIVE:	1. OFF	T	-	-	-	-	-	-	INACTIVE	
	INACTIVE	F	-	-	-	-	-	-	OFF	
	2. OFF	T	-	-	-	-	-	-	INACTIVE	
	INACTIVE	t	t	-	f	T	-	-	CRUISE	
	3. OVERRIDE	t	F	-	-	-	-	-	INACTIVE	
	INACTIVE	F	-	-	-	-	-	-	OFF	
	4. OVERRIDE	t	F	-	-	-	-	-	INACTIVE	
	INACTIVE	t	t	-	f	T	-	-	CRUISE	
	5. CRUISE	t	F	-	-	-	-	-	INACTIVE	
	OR									
	CRUISE	t	-	T	-	-	-	-	INACTIVE	
	INACTIVE	F	-	-	-	-	-	-	OFF	
	6. CRUISE	t	F	-	-	-	-	-	INACTIVE	
	OR									
	CRUISE	t	-	T	-	-	-	-	INACTIVE	
	INACTIVE	t	t	-	f	T	-	-	CRUISE	
	CRUISE:	1. INACTIVE	t	t	-	f	T	-	-	CRUISE
		CRUISE	F	-	-	-	-	-	-	OFF
2. INACTIVE		t	t	-	f	T	-	-	CRUISE	
CRUISE		t	F	-	-	-	-	-	INACTIVE	
OR										
CRUISE		t	-	T	-	-	-	-	INACTIVE	
3. INACTIVE		t	t	-	f	T	-	-	CRUISE	
CRUISE		t	t	f	T	-	-	-	OVERRIDE	
OR										
CRUISE		t	t	f	-	-	T	-	OVERRIDE	
4. OVERRIDE		t	t	-	f	T	-	-	CRUISE	
OR										
OVERRIDE		t	t	-	f	-	-	T	CRUISE	
CRUISE		F	-	-	-	-	-	-	OFF	
5. OVERRIDE		t	t	-	f	T	-	-	CRUISE	
OR										
OVERRIDE		t	t	-	f	-	-	T	CRUISE	
CRUISE		t	F	-	-	-	-	-	INACTIVE	
OR										
CRUISE		t	-	T	-	-	-	-	INACTIVE	

6. OVERRIDE t t - f T - - CRUISE
OR
 OVERRIDE t t - f - - T CRUISE
 CRUISE t t f T - - - OVERRIDE
OR
 CRUISE t t f - - T - OVERRIDE

OVERRIDE: 1. CRUISE t t f T - - - OVERRIDE
OR
 CRUISE t t f - - T - OVERRIDE
 OVERRIDE F - - - - - OFF
 2. CRUISE t t f T - - - OVERRIDE
OR
 CRUISE t t f - - T - OVERRIDE
 OVERRIDE t F - - - - - INACTIVE
 3. CRUISE t t f T - - - OVERRIDE
OR
 CRUISE t t f - - T - OVERRIDE
 OVERRIDE t t - f T - - CRUISE
OR
 OVERRIDE t t - f - - T CRUISE
