SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2003; **13**:25–53 (DOI: 10.1002/stvr.264)

Generating test data from state-based specifications

Jeff Offutt[1,*,†], Shaoying Liu[2], Aynur Abdurazik[1] and Paul Ammann[1]

[1]*ISE Department, George Mason University, Fairfax, VA 22030, U.S.A.*
[2]*Faculty of Information Sciences, Hosei University, 3-7-2 Kajino-cho Koganei-shi, Tokyo 184-8584, Japan*

SUMMARY

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 automatically 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 (transition predicates, transitions, pairs of transitions and sequences of transitions). 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. These criteria have been applied to a case study to compare their ability to detect seeded faults. Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: formal methods; specification-based testing; software testing

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

*Correspondence to: Jeff Offutt, Information & Software Engineering Department, George Mason University, Fairfax, VA 22030-4444, U.S.A.
†E-mail: ofut@ise.gmu.edu

Contract/grant sponsor: Rockwell Collins, Inc., U.S. National Science Foundation; contract/grant number: CCR-98-04111
Contract/grant sponsor: Ministry of Education, Culture, Sports, Science and Technology of Japan; contract/grant number: Grant-in-Aid for Scientific Research on Priority Areas (No. 14019081)

Received 29 April 2002
Accepted 5 February 2003

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 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 state-based 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. These criteria are meant to be general enough to be applied to a variety of specification languages that use a state-based representation. They have been applied to Software Cost Reduction (SCR) [1,2], CoRE [3], Unified Modelling Language (UML) Statecharts [4,5] and the Structured Object-oriented Formal Language (SOFL) [6,7]. The test data generation model includes techniques for generating tests at several levels of detail, with views moving from clauses in predicates on transitions, to single transitions, to pairs of transitions and finally to sequences of transitions. 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. The computer C executes T on P to create the *actual output*, which must be compared with the *expected output*. The expected output is produced with some knowledge of the specification. Thus, code-based generation uses the specification 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. The arc from S to P is labelled 'refine' because if specifications are formal, a refinement process may be used.

Specification-based test data generation has several advantages, particularly when compared with code-based generation. Tests can be created earlier in the development process and be ready for execution *before* the program is finished. Thus, testing activities can be shifted to an earlier part of the development process, allowing for more effective planning and utilization of resources. Additionally, the process of generating tests from the specifications will often help the test engineer 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 to S). Another advantage is that the essential part of the test data can be independent of any particular implementation of the specifications. Requirements/specifications can also be used as a basis for output checking, significantly reducing one of the major costs of testing. The arcs from S to T and from S to *Expected Output* are labelled with a '?', because these are currently active areas of research.

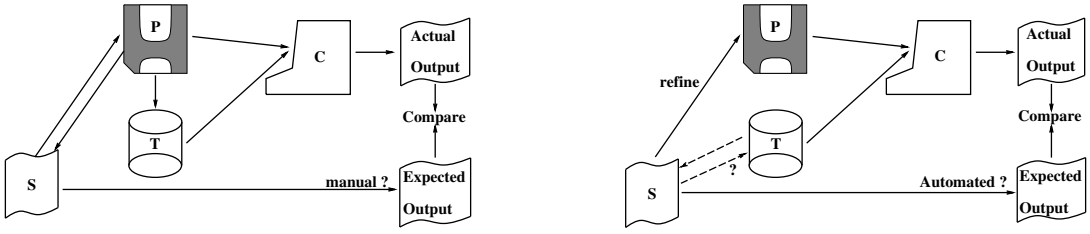


Figure 1. (a) Code-based test generation and (b) specification-based test generation.

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 [8–10], and as a basis for formalizing *test specifications* (as opposed to functional specifications) [11–13]. This paper is primarily concerned with the first use, that of generating test cases from specifications, commonly referred to as specification-based testing.

Specification-based testing is currently immature, which means formalized criteria and automated tool support are scarce. 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 the 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.

Code-based and specification-based 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. 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. One way to differentiate specification-based testing from code-based testing is to consider the questions that are being posed. Specification-based testing can be thought of as addressing the question ‘are the functionalities correct?’, whereas code-based testing addresses the question of ‘how much software is being covered during testing?’.

At the same time, both code-based testing and specification-based testing often use coverage criteria and some sort of representation of the system. Moreover, these coverage criteria are often identical or at least similar, including covering nodes in graphs or exercising clauses in predicates. At an abstract level, these are very similar for both code-based and specification-based testing; the difference is primarily in where the structures (graphs, clauses, etc.) come from.

This paper first presents criteria for generating tests from state-based specifications. Applications of these criteria to three different specification languages are discussed; examples are presented using SCR specifications [1,2] and CoRE [3] and a case study is used to illustrate the test derivation process and to evaluate these criteria in terms of their ability to detect seeded faults. 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 [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 percentage of requirements that are satisfied.

This paper presents 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, mathematically-based points at which to stop testing. These criteria assume that the software's functionality is described in terms of states and transitions (that is, not algebraic or model-based specifications). 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 [1,2] calls these WHEN conditions and triggering events. The values the triggering events have before the transition are sometimes called *before-values* and the values after the transition are sometimes called *after-values*. The state immediately preceding the transition is the *pre-state* and the state after the transition is the *post-state*.

Figure 2 illustrates this model with a simple transition that opens an elevator door. If the elevator button is pressed (the trigger event), the door opens only if the elevator is not moving (the precondition $elevSpeed = 0$).

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 test specifications are described in terms of simple algebraic predicates. The predicates use variables that appear in the specifications and operators that are legal on these variables (primarily equality, assignment and

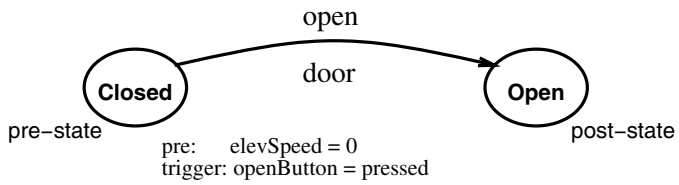


Figure 2. Elevator door open transition.

relational operators). The multi-level aspect means that tests are generated to test the software at several levels of abstraction on the specifications.

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* may be needed to terminate execution in some programs. *Expected outputs* are created from the after-values of the triggering events and any postconditions that are associated with the transition.

This paper defines four different test criteria at different levels of abstraction on the specifications, each of which requires a different amount of testing: (1) transition coverage; (2) full predicate coverage; (3) transition-pair coverage; and (4) complete sequence. These are defined in the following four sections. 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. Many state-based specification languages are fairly easy to translate into a specification graph as they have natural graph representations. Unfortunately, model-based specification languages such as Z are difficult to use in this way, in part because the state space would be very large. Other researchers have proposed ways to develop tests from Z specifications [13,15–23].

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 the minimum a tester should test is 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 include tests that cause every transition in the SG to be taken.

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 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. This paper restricts itself to Boolean logic using the operators AND ($\wedge$), OR ($\vee$) and NOT ($\neg$). There are straightforward ways to extend this analysis to other operators, including exclusive-OR ($\oplus$), implication ($\rightarrow$) and equivalence ($\leftrightarrow$), and any other Boolean algebras. This paper defines predicates and clauses 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 mathematics texts, DO-178B [24] 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. This is similar to the code-based testing criterion of modified condition/decision coverage (MC/DC) [25], as described in Section 2.5. Before defining full predicate coverage, it is necessary to clarify this notion of 'independently affecting the value of the predicate'. This is done with the notion of a clause 'determining' the value of the predicate.

Determination. Given a test clause c_i in predicate p, it is said that c_i *determines* p if the remaining *minor* clauses $c_j \in p$, $j \neq i$ have values so that changing the truth value of c_i changes the truth value of p.

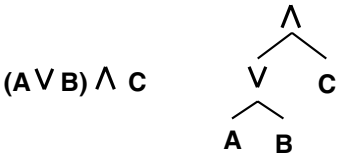
Note that this definition explicitly does *not* require that $c_i = p$. Some papers in the literature require the test clause and the predicate to have the same value and many leave the question ambiguous, which can lead to confusion when implementing criteria for logical expressions. This definition also imposes constraints on all clauses in the predicate, not just the test clause.

Full predicate coverage. For each predicate P on each transition and each test clause c_i in P, T must include tests that cause each clause c_i in P to *determine* the value of P, where c_i has both the values true and false.

Note that the use of the 'determines' definition implies that P will also have the values true and false. That is, if full predicate coverage is achieved, transition coverage will also be achieved. To satisfy the 'determination' requirement for the test clause, the other minor clauses must have specific values. For example, 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*

There are several ways to find values that satisfy full predicate coverage, including Aker's Boolean derivative method [26], as adapted by Kuhn [27], and Chilenski and Miller's pairs table method [25]. This paper presents a prescriptive approach that uses 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 `True`, then both children must have the value `True`; if an `and` node has the value of `False`, then at least one 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 at least one 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 3 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 in 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

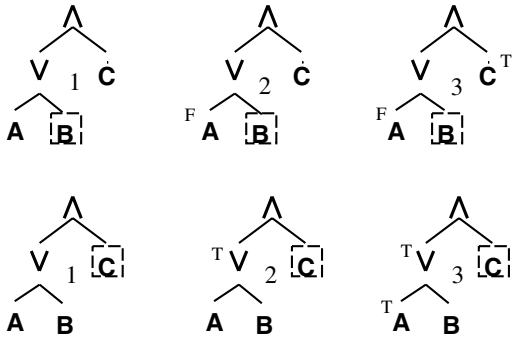


Figure 3. Constructing test case requirements from an expression parse tree.

requirement that the test clause must determine 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	**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 2, a triggering event is a change in a value for a variable, expression, or expressions that causes the software to undergo a 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.

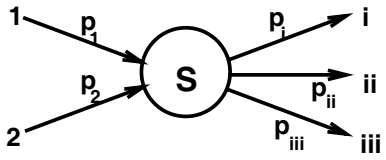
2.3. Transition-pair coverage criterion

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 types of faults may not be

adequately tested. Some faults may occur because an invalid sequence of transitions is allowed, or a valid sequence is not allowed. To check for these types 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 each transition of the pair in sequence.

Transition-pair subsumes transition coverage. An example of a transition-pair is illustrated by the following state:



To test the state S at the transition-pair criterion six tests are required, from: (1) 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 ordered 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. In particular, at the system level, effective testing probably requires detailed domain knowledge. The complete sequence criterion is not automatable or measurable like the other three criteria and this idea is certainly not new, but it is included here as an attempt at completeness. The definition is formulated so that this idea can be put into the same form and terminology as the previous criteria. 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 UML. 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.

Although the tests are intended to be executed on an implementation of the specification, a test is said to *traverse* a transition to indicate that, from a modelling 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.

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 only depend 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 DO-178B [24] and the definition is similar to that of MC/DC [25], 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 affect its decision independently. Although similar, the full predicate criterion is more restrictive and allows the satisfying procedure defined in Section 2.2.1. It should be emphasized, however, that the notion of coverage for the criteria in this paper is based on the specifications and there is no guarantee of code coverage. These criteria are designed to provide a range of test strengths; thus a contribution of this paper is to provide a practical range of cost/benefit choices for test engineers to construct tests from specifications.

3. AUTOMATICALLY DERIVING TEST CASES

Figure 4 illustrates the overall process used by a proof-of-concept tool built at George Mason University, SPECTEST. SPECTEST is implemented in Java and parses specifications into a general format (the specification graph), then generates test requirements for the appropriate criterion or criteria and then generates the actual test values. SPECTEST currently has parsers for SCR specifications created using the SCRTool developed at the Naval Research Laboratory [28], and UML Statecharts created by using Rational Software Corporation's Rational Rose tool [4].

The step of translating test specification values into program inputs (`Script Generator` in Figure 4) is not included in the SPECTEST tool. Testers need to relate variables that appear in the specifications to program variables or inputs. The simplest way is to assume that the variable names are the same (called 'testable software specifications' by Binder [29]), but a more general approach is to require that some sort of *mapping function* from specification variables to program variables exists. While this is a challenging and important problem, solving it would not contribute to the current research results and this is left as future research. The dashed line encloses the test generation steps that SPECTEST implements.

The following list describes each step that SPECTEST follows. Since some of the process steps are identical for each of the four criteria, they are presented together when possible. The criteria are differentiated after step 2.

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, as shown in the four `Spec Analyzer` boxes in Figure 4. 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.

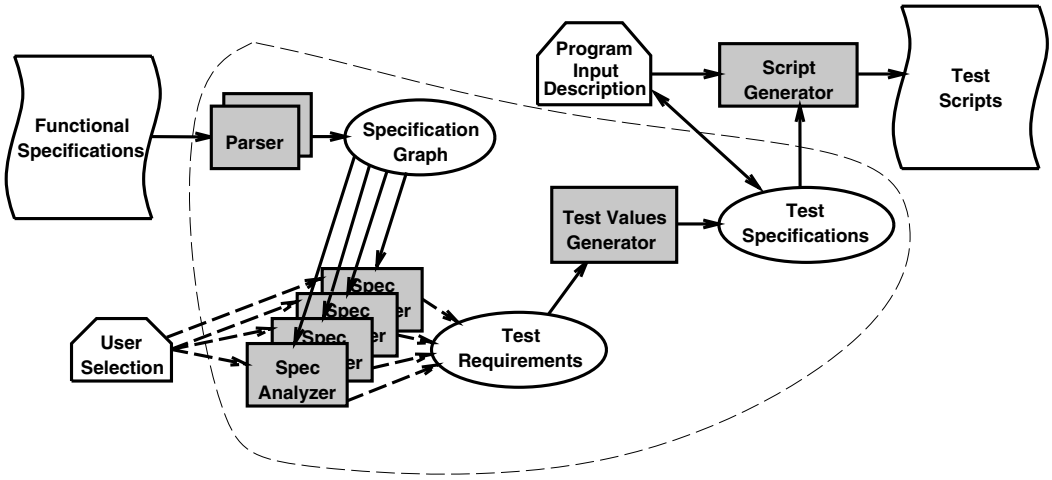


Figure 4. SPECTEST: general process for generating test cases.

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 specially marked so that the trigger event values appear after the precondition inputs (that is, they are ordered).

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 $M \times N$ 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. Although not implemented in SPECTEST, this could be 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 involves 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. *Script generator.* Each test specification is used to construct one test script. The actual scripts must reflect the input syntax of the program and have not been automated in SPECTEST. The script generator needs information about the inputs to the program. Specifically, it must solve the mapping problem and convert test specifications that are in terms of the specifications to executable test scripts that the software understands. Note that this is the only step that requires any knowledge of the implementation; all preceding steps depend solely on the functional specifications.

SPECTEST currently has the restriction that it processes only one mode class for SCR specifications at a time and one statechart class for UML specifications [5]. Although this imposes some burden on the tool user, in practice it makes no functional difference because testers usually test one mode or statechart class at a time. The tool is being expanded to handle other UML diagrams, most currently including collaboration diagrams.

4. CASE STUDY

As a preliminary demonstration of the feasibility of these criteria, an empirical study has been undertaken. A full experiment is planned for the future. The goal was to demonstrate that the specification-based criteria can be effectively used; the authors hope 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 in this study is described, the faults that were generated are listed and, finally, the results and analysis are given.

SPECTEST was used to generate test specifications from SCR mode tables for full predicate and transition-pair tests. For this implementation, the specification variables and program variables are the same and inputs are pairs of variables and values. This simplification means that the test specifications are also the test scripts. The tests include expected outputs (that is, the post-states); thus the output checking was done automatically.

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 branch testing and then the tests were measured in terms

Table I. SCR specifications for the cruise control system.

Previous mode	Ignited	Running	Toofast	Brake	Activate	Deactivate	Resume	New mode
Off	@T	—	—	—	—	—	—	Inactive
Inactive	@F	—	—	—	—	—	—	Off
Inactive	t	t	—	f	@T	—	—	Cruise
Cruise	@F	—	—	—	—	—	—	Off
Cruise	t	@F	—	—	—	—	—	Inactive
Cruise	t	—	@T	—	—	—	—	Inactive
Cruise	t	t	f	@T	—	—	—	Override
Cruise	t	t	f	—	—	@T	—	Override
Override	@F	—	—	—	—	—	—	Off
Override	t	@F	—	—	—	—	—	Inactive
Override	t	t	—	f	@T	—	—	Cruise
Override	t	t	—	f	—	—	@T	Cruise

of their fault-detection abilities. One moderately sized program was used, representative faults were seeded (by Offutt) and test cases were generated by hand (by Abdurazik).

Cruise control is a common example in the literature [1,30], and specifications are readily available. This case study used the SCR specification exactly as given by Atlee [1], shown in Table I. This version of cruise control does not model the throttle and has four states: OFF (the initial state), INACTIVE, CRUISE and OVERRIDE. Cruise is in INACTIVE if the engine is on (Ignited) and the cruise control has not been activated (Activate). Cruise can be activated only if the engine is on (Ignited), the engine is running (Running) and the brake is not currently being pressed (Brake). When the system is in CRUISE mode the computer controls the speed of the car. OVERRIDE mode is entered if the brake is pressed or the system is turned off.

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 for the speed to be controlled (*Toofast*), the brake pedal is being pressed (*Brake*) and whether the cruise control level is set to *Activate*, *Deactivate* or *Resume*.

Each row in the table specifies a conditioned event that activates a transition from the mode on the left to the mode on the right. A table entry of @T or @F under a column header C represents a triggering event @T(C) or @F(C). This means that the value of C must change for the transition to be taken; that is, '@T(C)' means C must change from false to true and '@F(C)' means C must change from true to false. A table entry of `t` or `f` represents a WHEN condition. WHEN[C] means the transition can only be taken if C is true and WHEN[¬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 '—' (do not care condition).

Table II. Expanded cruise control specification predicates.

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

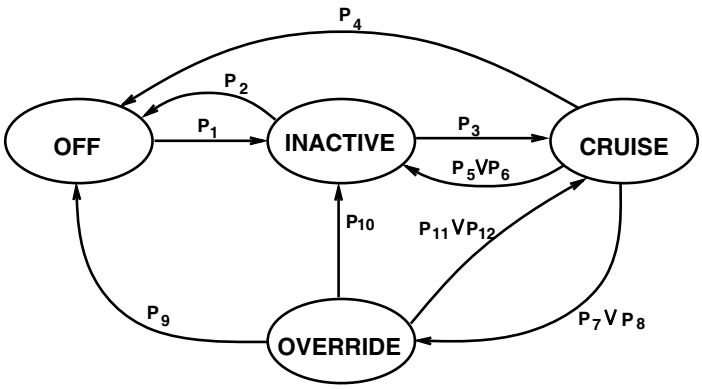


Figure 5. Specification graph for cruise control.

Table II shows the transitions of the specification with the trigger events expanded in predicate form, numbered P_1 through P_{12}. Figure 5 shows the specification graph, with edges labelled by the predicate numbers.

4.2. Test generation

The full predicate, transition, and transition-pair tests were generated by SPECTEST. The random tests referred to in Table III were generated by hand. To avoid bias, tests were created independently from the faults and by different people. 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 how tests were generated in detail.

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 the full predicate criterion, which are listed first for each transition.

4.2.1. Full predicate coverage criterion

There are nine transitions in the cruise control specifications and 12 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:

- $@T(X) \equiv NOT\ X \wedge X'$;
- $@T(X \wedge Y) \equiv \neg(X \wedge Y) \wedge (X' \wedge Y') \equiv (\neg X \vee \neg Y) \wedge X' \wedge Y'$;
- $@T(X \vee Y) \equiv \neg(X \vee Y) \wedge (X' \vee Y') \equiv \neg X \wedge \neg Y \wedge (X' \vee Y')$.

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 —. 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:

$@T\ Activate \wedge Ignited \wedge Running \wedge \neg Brake$

and expanding the triggering clause $@T\ Activate$ to include its before-value and after-value yields:

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

The six test case requirements for transition P_3 are:

Pre-state	*Activate*	*Ignited*	*Running*	*Brake*	*Activate′*	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 automatically derived from the test requirements. The predicate P_3 is chosen as an illustrative example. P_3 has six full predicate level tests. For the first test case for P_3, 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 P_3-1:

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

2. Test specification P_3-2:

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		

3. Test specification P_3-3:

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		

4. Test specification P_3-4:

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		

5. Test specification P_3-5:

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		

6. Test specification P_3-6:

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 that can decide what values need to be explicitly set are found in a technical report [31]; 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. The state-based specifications considered by this research have finite and deterministic specification graphs, so this problem is solvable for specification graphs derived from state-based systems. An algorithm for doing this is also in a technical report [31] 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.

The following are the test requirements for the four states.

OFF

1. P2 : P1
2. P4 : P1
3. P9 : P1

INACTIVE

1. P1 : P2
2. P1 : P3
3. P10 : P2
4. P10 : P3
5. (P5 OR P6) : P2
6. (P5 OR P6) : P3

CRUISE

1. P3 : P4
2. P3 : (P5 OR P6)
3. P3 : (P7 OR P8)
4. (P11 OR P12) : P4
5. (P11 OR P12) : (P5 OR P6)
6. (P11 OR P12) : (P7 OR P8)

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 II. 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 automatically 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 $P2$ and then $P1$.

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 branches. The program accepts pairs of variables and values, where a value can be 't', 'f', 'T' or 'F'. Uppercase 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 were in the logic that implemented the state machine. Four were naturally occurring faults, made during initial implementation. Faults were inserted by the first author.

4.4. Results and analysis

As a way to measure the quality of these tests, block and branch coverage was computed using the 54 full predicate test cases (created by the third author). The coverage was measured using Atac [32]. Five of the 174 branches were infeasible, leaving 169, and all of the blocks were feasible. The results are shown in Figure 6. The 54 test cases covered 163 of the blocks (89%) and 155 of the branches (95%). Of the 14 uncovered branches that were feasible, 11 were related to input parameters that were not used during testing. That is, these 11 branches were not related to the functional specifications. The remaining three branches were left uncovered because the variables *Activate*, *Deactivate* and

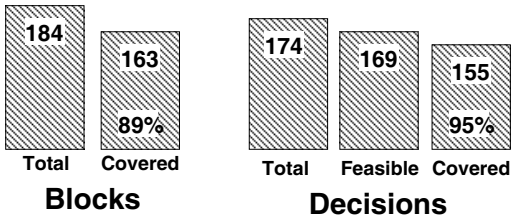


Figure 6. Branch and block coverage results.

Table III. Faults detected.

	Random	Transition	Transition-pair	Full predicate
Number of test cases	54	12	34	54
Faults found	15	15	18	20
Faults missed	9	9	6	4
Percentage coverage	62.5%	62.5%	75.0%	83.3%

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 transition coverage, an additional 42 for full predicate (making 54 total) and 34 were generated for the transition-pair criterion. 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 four sets of test data are shown in Table III.

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 four sets of tests found these three faults. The other fault that the full predicate tests missed was not found by either of the other three sets of inputs. The other two faults that were missed by the transition-pair tests were also missed by the transition and random tests. The other three faults

missed by the random and the transition tests were all different. All of the naturally occurring faults were found by all four 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 the 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 studies 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 [23,33–38], manual methods for deriving tests from specifications [15,16,20,39,40], case studies on using specifications to check the output of the software on specifications [11,22,41,42] and formalizations of test specifications [12,13,17,43]. 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. One of the earlier papers on model-based testing was by Zweben *et al.* [44]. They started with control-flow and data-flow testing techniques and applied them to abstract data types (ADTs). A graph was created where nodes represented functions from the ADT and an edge was created from $f1$ to $f2$ if $f2$ can be called after $f1$. The data flow analysis was based on the function pre- and post-conditions and criteria such as branch testing and all-uses were defined. A similar approach is applied in this paper, although the specification graph is very different. The specification graph defined in this paper is based on states in the software system behaviour, thus the tests are at the system level instead of for individual ADTs, as the tests from Zweben *et al.* were.

Spence and Meudec [45] and Dick and Faivre [40] suggested using specifications to produce predicates and then using predicate satisfaction techniques to generate test data. Given a set of predicates that reflect pre-conditions, invariants and post-conditions, 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 VDM pre- and post-conditions into disjunctive normal form predicates. Their paper did not fully address techniques for solving predicates, but suggested using Prolog theorem proving techniques and finite-state machines to sequence tests of operations. Although their work was based on completely different types of specifications (VDM), some of their ideas influenced the full predicate testing criterion in this paper. Their work does not include testing with invalid inputs, specifically

testing for faults, or comprehensive criteria for test selection and measurement. Derrick and Boiten [19] adapted Dick and Faivre's ideas to Z and proposed a solution for how to recreate tests when specifications change.

Stocks and Carrington [11,12], Amla and Ammann [15] and Ammann and Offutt [16] 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. This approach is based on a modification of the category-partition method for test generation [17,37]. Hierons [21] presented 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 [20] 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 [46] presented techniques to derive test conditions from an 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 *et al.* [35] 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 [41] used a similar approach to test object-oriented software.

Bernot [39] proposed a similar scheme, with more formalization of the process and the test cases. Bougé *et al.* [34] suggested a logic programming approach to generating test cases from algebraic specifications. Tsai *et al.* [23] 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 [47,48]. 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 to avoid brute force exploration of the state space.

Blackburn and Busser [49] used state-based functional specifications of the software, expressed in the language `T-Vec`, to derive disjunctive normal form constraints, similar 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 between Blackburn and Busser's algorithms and the algorithms used by Offutt's test data generator for mutation [50,51], the key difference being that Blackburn and Busser's is specification-based, whereas Offutt's constraints are code-based.

Weyuker *et al.* [38] 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 described here, but is restricted to Boolean variables.

5.4. Test generation via finite-state machines

Using finite state machines (FSM) for test generation has a long history. In the 1970s, Chow [52] used FSMs to generate tests for telecommunication systems. Arcs of the FSM are annotated with inputs and expected outputs. A spanning tree is generated from the FSM and test sequences are based on paths through this tree.

Test generation methods based on FSMs include tour [53], the W-method [52] and the partial-W method [54]. Their objective is to detect output errors based on state transitions driven by inputs. FSM-based test generation has been used to test a variety of types of applications including lexical analysers, real-time process control software, protocols, data processing and telephony.

FSMs have also been used to test object-oriented programs [55] and designs [56]. Kung *et al.* [55] extracted the FSM from the code using symbolic execution, while Turner and Robson [56] derived the FSM from the design of a class.

5.5. Summary

Most of the current specification-based testing techniques use manual methods that cannot be 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 based on 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 branch coverage and the technique was found to achieve a high level of coverage. It was also used to detect a large percentage of faults successfully. These results indicate that this technique can benefit software developers who construct formal specifications during development.

Although these criteria are introduced as being general and independent of language, both the specification language and the implementation language used have subtle impacts on how the criteria are interpreted and applied. To as much an extent as possible, test specifications are described in terms of simple algebraic predicates and the specification graph is developed as a language-independent intermediate representation. The predicates use variables that appear in the specifications and operators that are legal on these variables (primarily equality, assignment and relational operators).

Most languages have some form of algebraic predicate representation. The SPECTEST tool needs parsers to translate from the specification languages to the specification graph, a model that has been used successfully for three different specification languages.

One interesting result from the branch 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 branches having to do with the input were left out. For the 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.

APPENDIX A: 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 values 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.

		Variable values								
	Pre-state	I	R	T	B	A	D	S	Triggering event	Post-state
P1	OFF	F	—	—	—	—	—	—	*Ignited*′ = True	INACTIVE
	OFF	T	—	—	—	—	—	—	*Ignited*′ = True	OFF
	OFF	F	—	—	—	—	—	—	*Ignited*′ = False	OFF
P2	INACTIVE	T	—	—	—	—	—	—	*Ignited*′ = False	OFF
	INACTIVE	F	—	—	—	—	—	—	*Ignited*′ = False	INACTIVE
	INACTIVE	T	—	—	—	—	—	—	*Ignited*′ = True	INACTIVE
P3	INACTIVE	t	t	—	f	F	—	—	*Activate*′ = True	CRUISE
	INACTIVE	f	t	—	f	F	—	—	*Activate*′ = True	INACTIVE
	INACTIVE	t	f	—	f	F	—	—	*Activate*′ = True	INACTIVE
	INACTIVE	t	t	—	t	F	—	—	*Activate*′ = True	INACTIVE
	INACTIVE	t	t	—	f	T	—	—	*Activate*′ = True	INACTIVE
	INACTIVE	t	t	—	f	F	—	—	*Activate*′ = False	INACTIVE
P4	CRUISE	T	—	—	—	—	—	—	*Ignited*′ = False	OFF
	CRUISE	F	—	—	—	—	—	—	*Ignited*′ = False	CRUISE
	CRUISE	T	—	—	—	—	—	—	*Ignited*′ = True	CRUISE

		Variable values								
	Pre-state	I	R	T	B	A	D	S	Triggering event	Post-state
P5	CRUISE	t	T	—	—	—	—	—	$Running' = \text{False}$	INACTIVE
	CRUISE	f	T	—	—	—	—	—	$Running' = \text{False}$	CRUISE
	CRUISE	t	F	—	—	—	—	—	$Running' = \text{False}$	CRUISE
	CRUISE	t	T	—	—	—	—	—	$Running' = \text{True}$	CRUISE
P6	CRUISE	t	—	F	—	—	—	—	$Toofast' = \text{True}$	INACTIVE
	CRUISE	f	—	F	—	—	—	—	$Toofast' = \text{True}$	CRUISE
	CRUISE	t	—	T	—	—	—	—	$Toofast' = \text{True}$	CRUISE
	CRUISE	t	—	F	—	—	—	—	$Toofast' = \text{False}$	CRUISE
P7	CRUISE	t	t	f	F	—	—	—	$Brake' = \text{True}$	OVERRIDE
	CRUISE	f	t	f	F	—	—	—	$Brake' = \text{True}$	CRUISE
	CRUISE	t	f	f	F	—	—	—	$Brake' = \text{True}$	CRUISE
	CRUISE	t	t	t	F	—	—	—	$Brake' = \text{True}$	CRUISE
	CRUISE	t	t	f	T	—	—	—	$Brake' = \text{True}$	CRUISE
	CRUISE	t	t	f	F	—	—	—	$Brake' = \text{False}$	CRUISE
P8	CRUISE	t	t	f	—	—	F	—	$Deactivate' = \text{True}$	OVERRIDE
	CRUISE	f	t	f	—	—	F	—	$Deactivate' = \text{True}$	CRUISE
	CRUISE	t	f	f	—	—	F	—	$Deactivate' = \text{True}$	CRUISE
	CRUISE	t	t	t	—	—	F	—	$Deactivate' = \text{True}$	CRUISE
	CRUISE	t	t	f	—	—	T	—	$Deactivate' = \text{True}$	CRUISE
	CRUISE	t	t	f	—	—	F	—	$Deactivate' = \text{False}$	CRUISE
P9	OVERRIDE	T	—	—	—	—	—	—	$Ignited' = \text{False}$	OFF
	OVERRIDE	F	—	—	—	—	—	—	$Ignited' = \text{False}$	OVERRIDE
	OVERRIDE	T	—	—	—	—	—	—	$Ignited' = \text{True}$	OVERRIDE
P10	OVERRIDE	t	T	—	—	—	—	—	$Running' = \text{False}$	INACTIVE
	OVERRIDE	f	T	—	—	—	—	—	$Running' = \text{False}$	OVERRIDE
	OVERRIDE	t	F	—	—	—	—	—	$Running' = \text{False}$	OVERRIDE
	OVERRIDE	t	T	—	—	—	—	—	$Running' = \text{True}$	OVERRIDE
P11	OVERRIDE	t	t	—	f	F	—	—	$Activate' = \text{True}$	CRUISE
	OVERRIDE	f	t	—	f	F	—	—	$Activate' = \text{True}$	OVERRIDE
	OVERRIDE	t	f	—	f	F	—	—	$Activate' = \text{True}$	OVERRIDE
	OVERRIDE	t	t	—	t	F	—	—	$Activate' = \text{True}$	OVERRIDE
	OVERRIDE	t	t	—	f	T	—	—	$Activate' = \text{True}$	OVERRIDE
	OVERRIDE	t	t	—	f	F	—	—	$Activate' = \text{False}$	OVERRIDE
P12	OVERRIDE	t	t	—	f	—	—	F	$Resume' = \text{True}$	CRUISE
	OVERRIDE	f	t	—	f	—	—	F	$Resume' = \text{True}$	OVERRIDE
	OVERRIDE	t	f	—	f	—	—	F	$Resume' = \text{True}$	OVERRIDE
	OVERRIDE	t	t	—	t	—	—	F	$Resume' = \text{True}$	OVERRIDE
	OVERRIDE	t	t	—	f	—	—	T	$Resume' = \text{True}$	OVERRIDE
	OVERRIDE	t	t	—	f	—	—	F	$Resume' = \text{False}$	OVERRIDE

APPENDIX B: 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								

			I	R	T	B	A	D	S	
		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

ACKNOWLEDGEMENTS

This 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, Culture, Sports, Science and Technology of Japan under a Grant-in-Aid for Scientific Research on Priority Areas (No. 14019081).

REFERENCES

1. Atlee JM, Gannon J. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering* 1993; **19**(1):24–40.
2. Henninger K. Specifying software requirements for complex systems: New techniques and their applications. *IEEE Transactions on Software Engineering* 1980; **6**(1):2–12.
3. Faulk S, Brackett J, Ward P, Kirby J. The CoRE method for real-time requirements. *IEEE Software* 1992; **9**(5):22–33.
4. Rational Software Corporation. *Rational Rose 98: Using Rational Rose*. Rational Rose Corporation: Cupertino, CA, 1998.
5. Offutt J, Abdurazik A. Generating tests from UML specifications. *Proceedings of the Second International Conference on the Unified Modeling Language (UML '99)* Fort Collins, CO, October 1999 (*Springer Lecture Notes in Computer Science*, vol. 1723), 416–429.
6. Liu S, Offutt AJ, Ho-Stuart C, Sun Y, Ohba M. SOFL: A formal engineering methodology for industrial applications. *IEEE Transactions on Software Engineering* 1998; **24**(1):337–344. Special Issue on Formal Methods.
7. Offutt J, Liu S. Generating test data from SOFL specifications. *The Journal of Systems and Software* 1999; **49**(1):49–62.
8. Hlady M, Kovacevic R, Li JJ, Pekilis BR, Prairie D, Savor T, Seviora RE, Simser D, Vorobiev A. An approach to automatic detection of software failures. *Proceedings IEEE 6th International Symposium on Software Reliability Engineering (ISSRE)*, Toulouse, France, October 1995. IEEE Computer Society Press: Los Alamitos, CA, 1995; 314–323.
9. Li JJ, Seviora RE. Automatic failure detection with conditional-belief supervisors. *Proceedings IEEE 7th International Symposium on Software Reliability Engineering (ISSRE 96)*, White Plains, NY, October 1996. IEEE Computer Society Press: Los Alamitos, CA, 1996; 4–13.
10. Luqi, Yang H, Zhang X. Constructing an automated testing oracle: An effort to produce reliable software. *Proceedings IEEE Conference on Computer Software and Applications (COMPSAC)*, Taipei, Taiwan, November 1994. IEEE Computer Society Press: Los Alamitos, CA, 1994; 228–233.
11. Stocks P, Carrington D. The ISDM case study: A dependency management system (specification-based testing). *Technical Report*, The University of Queensland, Department of Computer Science, 1993.

12. Stocks P, Carrington D. Test Templates: A specification-based testing framework. *Proceedings 15th International Conference on Software Engineering*, Baltimore, MD, May 1993. IEEE Computer Society Press: Los Alamitos, CA, 1993; 405–414.
13. Stocks P, Carrington D. A framework for specification-based testing. *IEEE Transactions on Software Engineering* 1996; **22**(11):777–793.
14. Frankl PG, Weyuker EJ. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering* 1988; **14**(10):1483–1498.
15. Amla N, Ammann P. Using Z specifications in category partition testing. *Proceedings Seventh Annual Conference on Computer Assurance (COMPASS 92)*, Gaithersburg, MD, June 1992. IEEE Computer Society Press: Los Alamitos, CA, 1992; 3–10.
16. Ammann P, Offutt AJ. Using formal methods to derive test frames in category-partition testing. *Proceedings Ninth Annual Conference on Computer Assurance (COMPASS 94)*, Gaithersburg, MD, June 1994. IEEE Computer Society Press: Los Alamitos, CA, 1994; 69–80.
17. Balcer M, Hasling W, Ostrand T. Automatic generation of test scripts from formal test specifications. *Proceedings of the Third Symposium on Software Testing, Analysis, and Verification*, Key West, FL, December 1989. ACM Press: New York, 1989; 210–218.
18. Carrington D, MacColl I, McDonald J, Murray L, Strooper P. From Object-Z specifications to ClassBench test suites. *Software Testing, Verification, and Reliability* 2000; **10**(2):111–137.
19. Derrick J, Boiten E. Testing refinements of state-based formal specifications. *Software Testing, Verification, and Reliability* 1999; **9**(1):27–50.
20. Hayes IJ. Specification directed module testing. *IEEE Transactions on Software Engineering* 1986; **12**(1):124–133.
21. Hierons RM. Testing from a Z specification. *Software Testing, Verification, and Reliability* 1997; **7**(1):19–33.
22. Laycock G. Formal specification and testing: A case study. *Software Testing, Verification, and Reliability* 1992; **2**(1):7–23.
23. Tsai WT, Volovik D, Keefe TF. Automated test case generation for programs specified by relational algebra queries. *IEEE Transactions on Software Engineering* 1990; **16**(3):316–324.
24. RTCA-DO-178B. Software considerations in airborne systems and equipment certification, December 1992. Radio Technical Commission for Aeronautics.
25. Chilenski JJ, Miller SP. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal* 1994; **9**(5):193–200.
26. Akers SB. On a theory of boolean functions. *Journal of the Society for Industrial and Applied Mathematics* 1959; **7**(4):487–498.
27. Kuhn DR. Fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engineering and Methodology* 1999; **8**(4):411–424.
28. Heitmeyer C, Kirby J, Labaw B. Tools for formal specification, verification, and validation of requirements. *Proceedings 1997 Annual Conference on Computer Assurance (COMPASS 97)*, Gaithersburg, MD, June 1997. IEEE Computer Society Press: Los Alamitos, CA, 1997; 35–47.
29. Binder RV. *Testing Object-oriented Systems*. Addison-Wesley Publishing Company Inc.: New York, 2000.
30. Jin Z. Deriving mode invariants from SCR specifications. *Proceedings Second IEEE International Conference on Engineering of Complex Computer Systems*, Montréal, Canada, October 1996. IEEE Computer Society Press: Los Alamitos, CA, 1996; 514–521.
31. Offutt AJ. 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/.
32. Horgan JR, London S. ATAC: A data flow coverage testing tool for C. *Proceedings Symposium of Quality Software Development Tools*, New Orleans, LA, May 1992. IEEE Computer Society Press, 1992; 2–10.
33. Bernot G, Gaudel MC, Marre B. Software testing based on formal specifications: A theory and a tool. *Software Engineering Journal* 1991; **6**(6):387–405.
34. Bougé L, Choquet N, Fribourg L, Gaudel M-C. Test sets generation from algebraic specifications using logic programming. *The Journal of Systems and Software* 1986; **6**(4):343–360.
35. Gannon J, McMullin P, Hamlet R. Data-abstraction implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems* 1981; **3**(3):211–223.
36. Jalote P. Specification and testing of abstract data types. *Computer Language* 1992; **17**(1):75–82.
37. Ostrand TJ, Sigal R, Weyuker EJ. Design for a tool to manage specification-based testing. *Proceedings of the Workshop on Software Testing*, Banff, Alberta, July 1986. IEEE Computer Society Press: Los Alamitos, CA, 1986; 41–50.
38. Weyuker E, Goradia T, Singh A. Automatically generating test data from a boolean specification. *IEEE Transactions on Software Engineering* 1994; **20**(5):353–363.
39. Bernot G. Testing against formal specifications: A theoretical view. *TAPSOFT'91: Proceedings of the International Joint Conference on Theory and Practice of Software Development*, vol. 2: Advances in Distributed Computing (ADC) and

Colloquium on Combining Paradigms for Software Development (CCPSD), Brighton, U.K., April 1991 (*Lecture Notes in Computer Science*). Springer: Berlin, 1991; 99–119.
40. Dick J, Faivre A. Automating the generation and sequencing of test cases from model-based specifications. *Proceedings of FME '93: Industrial-Strength Formal Methods*, Odense, Denmark (*Lecture Notes in Computer Science*, vol. 670). Springer: Berlin, 1993; 268–284.
41. Doong RK, Frankl PG. Case studies on testing object-oriented programs. *Proceedings Fourth Symposium on Software Testing, Analysis, and Verification*, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society Press: Los Alamitos, CA, 1991; 165–177.
42. Kemmerer RA. Testing formal specifications to detect design errors. *IEEE Transactions on Software Engineering* 1985; **11**(1):32–43.
43. Choquet N. Test data generation using a Prolog with constraints. *Proceedings Workshop on Software Testing*, Banff, Alberta, July 1986. IEEE Computer Society Press: Los Alamitos, CA, 1986; 51–60.
44. Zweben SH, Heym WD, Kimmich J. Systematic testing of data abstractions based on software specifications. *Software Testing, Verification, and Reliability* 1992; **1**(4):39–55.
45. Spence I, Meudec C. Generation of software tests from specifications. *Proceedings SQM'94*, vol. 2, Edinburgh, Scotland, July 1994. Computational Mechanics Publications, 1994; 517–530.
46. Chang J, Richardson D. Structural specification-based testing with ADL. *Proceedings of the 1996 International Symposium on Software Testing, and Analysis*, San Diego, CA, January 1996. ACM Press: New York, 1996; 62–70.
47. Ammann PE, Black PE. A specification-based coverage metric to evaluate test sets. *HASE 99: Fourth IEEE International Symposium on High Assurance Systems*, Washington, DC, November 1999. IEEE Computer Society Press: Los Alamitos, CA, 1999; 239–248.
48. Ammann PE, Black PE, Majurski W. Using model checking to generate tests from specifications. *Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, Brisbane, Australia, December 1998. IEEE Computer Society Press: Los Alamitos, CA, 1998; 46–54.
49. Blackburn M, Busser R. T-VEC: A tool for developing critical systems. *Proceedings 1996 Annual Conference on Computer Assurance (COMPASS 96)*, Gaithersburg, MD, June 1996. IEEE Computer Society Press: Los Alamitos, CA, 1996; 237–249.
50. DeMillo RA, Guindi DS, King KN, McCracken WM, Offutt AJ. An extended overview of the Mothra software testing environment. *Proceedings Second Workshop on Software Testing, Verification, and Analysis*, Banff, Alberta, July 1988. IEEE Computer Society Press: Los Alamitos, CA, 1988; 142–151.
51. DeMillo RA, Offutt AJ. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering* 1991; **17**(9):900–910.
52. Chow T. Testing software designs modeled by finite-state machines. *IEEE Transactions on Software Engineering* 1978; **4**(3):178–187.
53. Naito S, Tsunoyama M. Fault detection for sequential machines by transition tours. *Proceedings Fault Tolerant Computing Systems*, 1981; 238–243.
54. Fujiwara S, Bochman G, Khendek F, Amalou M, Ghedasmi A. Test selection based on finite state models. *IEEE Transactions on Software Engineering* 1991; **17**(6):591–603.
55. Kung D, Gao J, Hsia P, Toyoshima Y, Chen C. A test strategy for object-oriented programs. *19th Computer Software and Applications Conference (COMPSAC 95)*, Dallas, TX, August 1995. IEEE Computer Society Press, 1995; 239–244.
56. Turner CD, Robson DJ. The state-based testing of object-oriented programs. *Proceedings 1993 IEEE Conference on Software Maintenance (CSM-93)*, Montréal, Quebec, Canada, September 1993. IEEE Computer Society Press: Los Alamitos, CA, 1993; 302–310.

