Generating Test Data From Requirements/Specifications: Phase III Final Report

Prepared For: Rockwell Collins, Inc
Contract Monitor: Dave Statezni

Principal Investigator: A. Jefferson Offutt
George Mason University

August 29, 1999
Update November 24, 1999

EXECUTIVE SUMMARY

This report presents results for the Rockwell Collins Inc. sponsored project on generating test data from requirements/specifications, which started January 1, 1999. The purpose of this project is to improve our ability to test software that needs to be highly reliable by developing formal techniques for generating test cases from formal specificational descriptions of the software. 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 by automated means.

This Phase III, 1999 report presents results from an empirical evaluation of the full predicate specification-based testing criterion developed during the first two phases of this project, and a proof-of-concept test data generation tool. The evaluation used a comparative study on a large industrial system, a research version of the Flight Guidance Mode Logic System (FGS) provided by Rockwell Collins. Full predicate tests were generated for FGS, and compared against the T-Vec generation scheme. T-Vec tests for FGS were also provided by Rockwell Collins. While creating and running the tests, one problem was found in the SCR specifications for FGS, and one problem was found in the already well tested implementation of FGS. Both T-Vec and the full predicate tests found almost the same number of faults, but T-Vec required more than five times as many tests, thus the full predicate tests were more **efficient**. The proof-of-concept test data generator creates full predicate and transition-pair tests from an SCR specification. Currently, the tool requires the SCR specification to be a single mode transition table, all variables must be boolean, and the transition predicates must be single-variable expressions.

Technical Report ISE-TR-00-02, Department of Information and Software Engineering, George Mason University, Fairfax VA, May, 2000.

1 INTRODUCTION

Software system-level tests have traditionally been based on informal, ad-hoc analyses of the system requirements. This can lead to inconsistent results, problems in understanding the goals and results of testing, and an overall lack of effectiveness in testing. This research project is attempting to establish formal criteria and processes for generating system-level tests from functional requirements/specifications. The purpose is to improve our ability to test software that needs to be highly reliable.

Thus far, this work has resulted in a general model for developing test inputs from state-based specifications. This model includes several related criteria for generating test data from formal specifications. These criteria provide a formal process, a method for measuring tests, and a basis for full automation of test data generation.

The principal results in this report are from an empirical comparison of two specification-based test strategies. The full-predicate criteria from previous Phase I (during 1997) [Off98, OXL99] and Phase II (during 1998) [Off99] are compared with T-Vec generated tests [BB96, Bla98]. Also presented is a proof-of-concept automated test data generation tool. This tool implements the criteria and some of the algorithms that were developed in previous years. The report begins by summarizing the results from Phases I and II, then presents the current year goals.

2 SUMMARY OF PHASES I AND II

Phase I of this project was carried out during Summer 1997, and established the long term goal of improving our ability to test software that needs to be highly reliable by developing formal techniques for generating test cases from formal specificational descriptions of the software [Off98]. This research addressed the problem of developing formalizable, measurable criteria for generating test cases from specifications.

During Phase I, a general model for developing test inputs from state-based specifications was developed. This model includes a derivation process for obtaining the test cases, and the report included an example for a small system, and test cases from specifications of an industrial system. The test data generation model includes techniques for generating tests at several levels of abstraction for specifications, including the complete transition sequence level, the transition-pair level, the transition level, and the full predicate level. These techniques are novel in that they provide coverage criteria that are based on the specifications. It is thought that these are the first formal coverage criteria for functional specifications. The tests 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. A test generation process was also developed, which includes several steps for transforming specifications to tests.

Results from applying the model and process to a small example were presented in the Phase I final report. This case study was evaluated using Atac to measure decision coverage, and the technique was found to achieve a high level of coverage. This result indicates that this technique can benefit software developers who construct formal specifications during development.

As an additional validation, tests were generated for specifications of an industrial software system supplied by Rockwell-Collins, a research version of the Flight Guidance Mode Logic System. Construction of these tests resulted in several modifications to this technique, and found at least one problem with the specification.

Phase II of this project was carried out during Spring and Summer 1998. It resulted in algorithms for test case development, and a small empirical evaluation of the test criteria [Off99].

One significant problem in specification-based test data generation is that of reaching the proper program state necessary to execute a particular test case. Given a test case that must start in a particular state S, the test case *prefix* is a sequence of inputs that will put the software into state S. This problem was addressed in two ways. First is to combine various test cases to be run in *test sequences* that are *ordered* in such a way that each test case leaves the software in the state necessary to run the subsequent test case. An algorithm was developed that attempts to find test case sequences that are *optimal* in the sense that the fewest possible number of test cases are used. Second, to handle situations where it is desired to run each test case independently, an algorithm for directly deriving test sequences was created.

The final report for Phase II also presented procedures for removing redundant test case values, and developed the idea of "sequence-pair" testing into a more general idea of "interaction-pair" testing. A small case study was also carried out. This case study applied the test criteria of transition coverage and full predicate coverage to the well known cruise control example. The results were that the specification-based criteria covered most of the blocks and decisions in the program source code, and found a high percentage of faults that were inserted into the source code.

2.1 Summary of Phase III Goals

The current year research carries the previous results forward in two directions. The first results presented are from a formal **empirical evaluation** of the specification-based testing technique developed during the first two phases of this project. This evaluation was done by applying the testing technique to a research version of the Flight Guidance Mode Logic System (FGS) example supplied by Rockwell Collins. Rockwell Collins supplied an SCR specification of FGS that was based on Miller and Hoech's CoRE specifications [MH97], and a Java implementation. Two sets of data are presented. The first is based on the specification-based testing criteria developed by this project and the other is based on tests provided by Rockwell Collins. These tests were generated by T-Vec, a system level, automatic test data generator [BB96, Bla98].

The two sets of data were compared on the basis of fault finding effectiveness. To measure effectiveness, a number of faults were injected into the software, the tests were executed, and the number of faults found by both sets of tests are reported.

The second direction is in terms of automation. A preliminary **proof-of-concept automatic test data generator** has been developed. This tool applies the test data generation criteria from the first two phases of this project to automatically create inputs. There are currently two sources for the tests; SCR specifications and UML Statecharts. The SCR specifications are created by using the SCRTool developed at the Naval Research Laboratory [HKL97], and the UML Statecharts are created using Rational Software Corporation's Rational Rose tool [Cor98]. The tool reads the specifications in either the UML or the SCR format, then generates appropriate test cases. Initial results from using this tool have already been accepted for publication and will appear in the UML conference this fall [OA99].

2.2 Publications From This Project

Thus far, this project has resulted in the following publications. All publications acknowledge Rockwell Collins as complete or partial sponsor (related support has also been provided by the National Science Foundation and the Government of Japan). All publications (except the technical reports) are completely refereed. Two journal papers and two additional conference papers are also currently in preparation.

1. Jeff Offutt and Aynur Abdurazik. Generating Tests from UML Specifications. *Second International Conference on the Unified Modeling Language (UML '99)*, Fort Collins, CO, October 1999.

2. Jeff Offutt, Yiwei Xiong and Shaoying Liu. Criteria for Generating Specification-based Tests. *Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, pages 119-131, Las Vegas, NV, October 1999.

3. Zhenyi Jin and Jeff Offutt. Coupling-based Integration Testing. *Second IEEE International Conference on Engineering of Complex Computer Systems*, pages 10–17, Montreal, Canada, October 1996. (Outstanding Paper Award).

4. Jeff Offutt, Generating Test Data From Requirements/Specifications: Phase II Final Report, January 1999, George Mason University Department of ISE Technical Report ISSE-TR-99-01, http://www.ise.gmu.edu/techrep.

5. Jeff Offutt, Generating Test Data From Requirements/Specifications: Phase I Final Report, April 1998, George Mason University Department of ISE Technical Report ISSE-TR-98-01, http://www.ise.gmu.edu/techrep.

3 EMPIRICAL EVALUATION

Evaluating testing criteria is always difficult. If a particular criterion is evaluated in isolation, the question always remains as to whether the criterion was valuable, or the evaluation was weak. For example, suppose a criterion is evaluated in terms of how many faults it finds. If it finds a lot of faults, then it could be because the criterion is strong, or because the faults were easy to find. Thus, testing criteria are usually evaluated by use of **comparative** studies.

Comparing testing criteria is also problematic. Analytical comparisons (for example, proofs and subsumptive hierarchies) show theoretical relationships between techniques, and are most satisfying because they allow claims that are true in all situations. Unfortunately it is often impossible to make analytical comparisons. Empirical comparisons show relations that are based on specific studies. Although it is difficult to show that empirical results hold in all situations, analytical comparisons cannot always be made, but empirical comparisons can.

Two relationships that have been defined elsewhere are the ProbBetter and the ProbSubsumes relationships. Weyuker, Weiss, and Hamlet [WWH91] suggest a relationship called ProbBetter:

> A testing criterion C_1 is ProbBetter than C_2 for a program P if a randomly selected test set `T` that satisfies C_1 is more "likely" to detect a failure than a randomly selected test set that satisfies C_2.

Mathur and Wong [MW94, Won93] suggest a different relationship called ProbSubsumes:

> A testing criterion C_1 ProbSubsumes C_2 for a program P if a test set T that is adequate with respect to C_1 is "likely" to be adequate with respect to C_2. If C_1 ProbSubsumes C_2, C_1 is said to be more "difficult" to satisfy than C_2.

The ProbBetter relation is defined with respect to the fault detection capability of test sets, whereas the ProbSubsumes relation is defined with respect to the difficulty of satisfying one criterion in terms of another. To evaluate ProbSubsumes, the tests that are generated must be defined in terms of a test criterion. To evaluate ProbBetter, the tests must be executed on faulty versions of programs. Both are probabilistic relations between two testing criteria and are defined in terms of specific programs. Although this means that it is difficult to draw general conclusions from any one study, as the number or variety of programs studied increase, our confidence in the validity of a ProbSubsumes or a ProbBetter relationship with a larger set of programs also increases.

3.1 Experimental Hypotheses and Design

The specification-based testing criterion is being evaluated by using a comparative study. It is being compared with the T-Vec testing technique [BB96, Bla98]. T-Vec includes a technique for generating test data from software specifications, but does not include a formal criterion, thus the ProbSubsumes relationship cannot be used. The T-Vec test cases were supplied by Rockwell-Collins, and compared with full predicate (FP) test cases on the basis of the ProbBetter relationship. For this comparison, the following hypotheses have been formulated:

Hypothesis 1: Full predicate testing is ProbBetter than T-Vec testing.
Hypothesis 2: T-Vec testing is ProbBetter than full predicate testing.

As an experimental subject, a research version of the Flight Guidance Mode Logic System (FGS) supplied by Rockwell Collins was used [MH97]. FGS is large enough to provide results that are meaningful to industry software developers, and formal specifications already exist. A number of faults were injected into the FGS software, and the number of faults found by both sets of tests was tracked.

A *minimum* test case set for a criterion contains the smallest number of test cases necessary to satisfy the criterion, and a *minimal* test case set is a satisfying set such that if any test case was removed, the set would no longer satisfy the criterion. Finding a minimum test set requires an exponential search process (an NP-complete problem), so this experiment uses minimal test sets. Redundant test cases were eliminated using previously created test set reduction algorithms.

3.2 Experimental Conduct

The following process was used during the experiment:

1. Acquire FGS specifications in SCR
2. Acquire FGS implementation from Rockwell
3. Acquire Rockwell-supplied T-Vec tests
4. Create full predicate tests
5. Insert faults into FGS
6. Run every test case, gather and interpret results

The order of these steps is important. The specific values for both test sets could be influenced by knowledge of the faults, so the tests had to be created first. In a system the size of FGS, knowledge of tests that are based on specifications are very unlikely to influence faults that are inserted into the code, so the faults can be created after the tests.

3.2.1 Flight guidance Mode Logic system

An SCR specification for FGS and a Java implementation of FGS were provided by Rockwell Collins. The SCR specification was derived from Miller and Hoech's CoRE specifications [MH97]. The specification contains 14 mode transition tables and 36 supporting tables (condition tables, term tables, etc.). The implementation contains 115 Java classes and about 6500 lines. This was a research version of the implementation, and was provided without a user interface or other main program.

It was decided to derive tests for the mode transition tables, which describe the most visible parts of the system. These tables describe the conditions under which the various objects in the FGS change their modes. There are 14 mode transition tables in the SCR specifications, and they reference variables and terms that are defined in other tables.

3.2.2 T-Vec tests

The T-Vec tests were also provided by Rockwell in the form of Java test driver programs with test data embedded in the drivers. A total of 3732 tests were embedded in 14 drivers. One driver, for the Flight Modes Flight Director Mode transition table, would not compile as given. Flight_Modes_Flight_Director_Mode_Test_Driver.java contained over 300,000 lines of Java code, including 2763 tests, and the Java compiler (Sun JDK 1.2.10) ran out of memory when compiling it. To run this driver, it was broken into 12 separate drivers, each containing 250 tests (the last only contained 14 tests).

A goal of T-Vec is to generate tests that provide MCDC structural coverage [CM94] on the source code implementation. In fact, it might be MCC, which is even more extensive. T-Vec did not try to reduce the number of tests in any way.

3.2.3 Full predicate tests

Although a proof-of-concept test data generation tool exists for full predicate tests, it is not complete and cannot yet handle many of the structures used in the FGS SCR specifications. The full predicate tests were generated partially by the tool, and partially by hand. A total of 735 tests were generated from the 14 mode transition tables.

During test generation, several issues related to generating full predicate testing had to be resolved. The first was with the "@C (A)" predicate. In SCR, @T indicates that the predicate must change from false to true, @F indicates that the predicate must change from true to false, and @C simply indicates that the predicate must change. In the full predicate test generation strategy, @T operators are expanded as:

`@T(A)` $\equiv \neg$ `A AND A'`

where A' is the value of the predicate A after the change. The equivalent expansion for @C had not been defined, and three alternatives presented themselves:

```
1) @C(A) ≡ A ≠ A'
2) @C(A) ≡ @T(A) ∨ @F(A)
3) @C(A) ⇒ i) @T(A)
           ii) @F(A)
```

The first expansion is simple, direct, and consistent with the other trigger operators ("@T" and "@F"). However, if the condition is a little complicated, generating tests can get messy. For example, if the trigger is: `@C(X <= Y)`, then the expansion is: `(X<=Y) != (X<=Y)'`, and there are many choices for how to satisfy the expression. When the event has multiple conditions, the situation is even more difficult. For example, if the predicate is: `@C(A AND B) == (A AND B) != (A AND B)'`, there are many ways to satisfy it. It is not clear if they should all be chosen, or only some of them. Consider a truth table approach:

```
    (A AND B) != (A ANDB)'
1)  T     T     T   T
2)  T     F     T   F
3)  F     T     F   T
4)  F     F     F   F
```

The triggering event could be satisfied by combining choice 1 on the left with options 2, 3, and 4 on the right, or by combining choices 2, 3, and 4 on the left with option 1 on the right:

```
T T -- T F
T T -- F T
T T -- F F
T F -- T T
F T -- T T
F F -- T T
```

These are a lot of choices, which makes it more difficult for the method to be applied by hand and more difficult to automate. If the above predicate is expanded even further, the following six disjunctive clauses are gotten:

```
@C(A AND B) == (A AND B) != (A AND B)'
            == (ABA' NOT B') OR (AB NOT A'B') OR (AB NOT A' NOT B') OR
               (A NOT BA'B') OR ( NOT ABA'B') OR ( NOT A NOT BA'B')
```

This would result in up to 30 test cases, which seems excessive for one trigger event. The second expansion is not as messy, but still leads to a large number of test cases.

```
@C(A) == @T(A) OR @F(A)
      == (A AND NOT A') OR (NOT A AND A')
```

If the trigger contains multiple conditions the fully expanded expression looks like:

```
@C(A AND B) == @T(A AND B) OR @F(A AND B)
            == ((NOT A OR NOT B) AND A' AND B') OR
               (A AND B AND (NOT A' OR NOT B'))
```

Semantically this is equivalent to the first expansion, but fewer test cases will result.

The third expansion allows a full separation, and allows both cases (false to true and true to false) to be tested. It is non-intuitive from a specification point of view, but might be effective from a testing point of view. That is, it could lead to effective tests at reasonable cost.

A careful analysis of the SCR specification, however, indicated that all @C() triggers used floating point expressions, and none used boolean expressions. In this case, the potential complexities in the first two expansions are not realized. In fact, the most likely use of @C triggers seems to be when a numeric quantity is changed as opposed to a boolean variable. Thus, we chose the first expansion for this research project.

The second issue was handling transition predicates that included values for mode transition tables instead of variables. For example, one transition was:

```
Current Mode:     CLEARED
Expected Mode:    CUES
Event:            @T(Flight_Modes_Flight_Director_Mode = ON)
```

The event term `Flight_Modes_Flight_Director_Mode` is also a mode transition table, so the expression could not be set directly. This was handled by a straightforward recursive substitution. The term `(Flight_Modes_Flight_Director_Mode = ON)` was replaced by the appropriate sequence of value assignments from the `Flight_Modes_Flight_Director_Mode` mode transition table to put that table into the mode ON.

3.2.4 Faults inserted into FGS

The faults were also inserted by hand. The strategy was to insert faults that are (1) similar to naturally occurring faults, and (2) not trivial to detect. All faults passed the Java compiler (Sun JDK 1.2.10). A total of 155 faults were created. A general outline of the fault creation procedure is that for each program statement, we attempted to:

1. transpose variables
2. modify multiple, related, arithmetic or relational operators
3. change precedence of operation (i.e., by changing parenthesis)
4. delete a conditional or iterative clause
5. change conditional expressions by adding extra conditions

6. change the initial values and stop conditions of iteration variables

To gather the results, each fault was inserted into a separate Java source file, creating 155 incorrect versions of FGS. This simplifies data gathering by making it clear which fault is detected when the faulty program fails. One complication from using this strategy with a Java implementation is that the Java compiler does not create complete executables, rather it compiles each Java source file to a `class` bytecode file, and the `class` files are then interpreted during execution. Thus, to execute the multiple faulty versions of FGS, each Java file was compiled to a faulty `class` file, and we developed a shell script that copied the faulty class file into the appropriate directory before running FGS. Although this was entirely automated, it was quite slow (for example, it took over eight hours to run the 735 full predicate test cases on all 155 faulty programs).

Figure 1 shows the directory structure of FGS and location of the faulty Java classes. All faulty classes are stored in subdirectories called `BuggyVersions/`. In Figure 1, each Java file is annotated with the number of faults created for it, and the `BuggyVersions/` directories are annotated with the total number of faults in the directory. No faults were placed into the `BusLayer`, `AircraftDSOLayer`, or the `ControlsDSOLayer` classes. This is because tests generated from the SCR mode transition tables could not cause methods in those classes. In addition, `CommonTypes` contains classes that are not directly described or modeled in the functional specifications. Faults were added for `CommonTypes` as a measure of how well the specification-based tests were able to test code that is not directly described by the specifications.

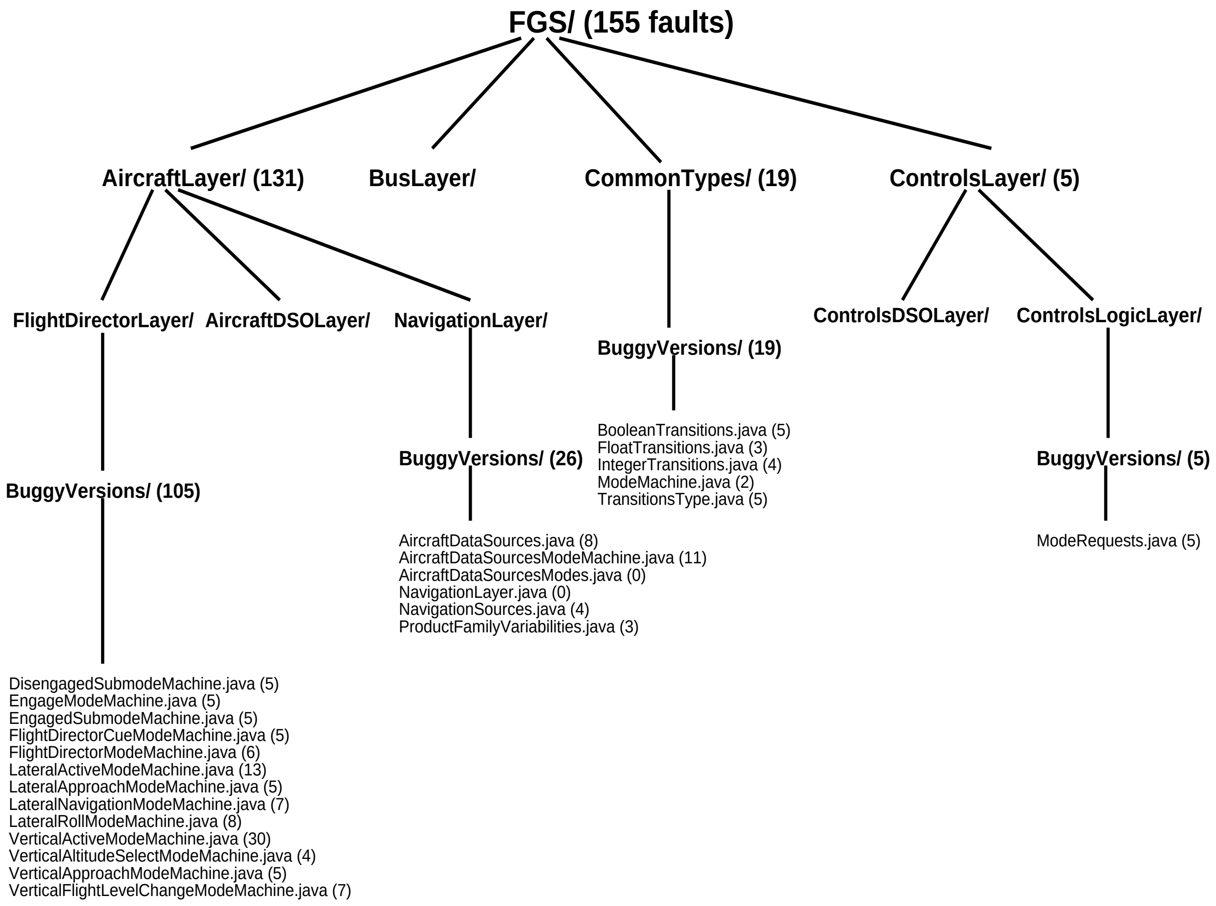
3.2.5 FGS test driver program

T-Vec tests were embedded in their own drivers, which were automatically generated by the T-Vec system. Rockwell Collins personnel created a *driver template file* and a set of *mappings tables*. For each variable that appeared in the SCR specification, a mappings table provided the methods within the implementation for setting and retrieving values for the variable. The driver template file and the mappings tables were fed into the T-Vec system, which produced a set of driver programs that contained tests.

Execution of the full predicate tests on FGS required the construction of a general purpose test driver. FGS was provided as a collection of Java classes without a main program, thus test drivers had to be created to run the tests. It was decided to create one general purpose driver that (1) reads in values for variables, (2) executes the appropriate methods to assign the values to the variables, (3) calls the appropriate `update()` methods to run the FGS system, and (4) prints the results (the new state). The appropriate methods are derived from the mapping files created by Dave Statezni for the T-Vec drivers. The test driver is implemented in two Java classes of about 750 lines of code, and is provided in Appendix B. Figure 2 provides the input specification for the driver. It includes comment lines, a command to set the current mode transition table, a command to set the previous mode, a command to explicitly update the current modes for the system, and commands to set values for variables. Note that no input validation is done. Results checking is done outside of the test driver. In the experiment, a Unix shell script was used to capture the outputs from the driver. When the original version of FGS was run, the outputs were compared against the expected outputs in the test case file to verify correctness, when faulty versions of FGS were run, the outputs were compared against the outputs from the original version by use of the Unix tool `diff`.

3.3 Problems Found During Test Preparation

While preparing the tests, we were able to uncover one fault in the FGS implementation, one problem with the SCR specification, and one problem with the mappings tables. When generating tests for the `Flight_Modes_Flight_Director_Mode` mode transition table, a fault in its implementation (`FlightDirectorModeMachine.java`) was found. The first transition for that table is:



FGS/ (155 faults)
AircraftLayer/ (131)
BusLayer/
CommonTypes/ (19)
ControlsLayer/ (5)
FlightDirectorLayer/
AircraftDSOLayer/
NavigationLayer/
BuggyVersions/ (19)
ControlsDSOLayer/
ControlsLogicLayer/
BooleanTransitions.java (5)
FloatTransitions.java (3)
IntegerTransitions.java (4)
ModeMachine.java (2)
TransitionsType.java (5)
BuggyVersions/ (26)
BuggyVersions/ (5)
BuggyVersions/ (105)
AircraftDataSources.java (8)
AircraftDataSourcesModeMachine.java (11)
AircraftDataSourcesModes.java (0)
NavigationLayer.java (0)
NavigationSources.java (4)
ProductFamilyVariabilities.java (3)
ModeRequests.java (5)
DisengagedSubmodeMachine.java (5)
EngageModeMachine.java (5)
EngagedSubmodeMachine.java (5)
FlightDirectorCueModeMachine.java (5)
FlightDirectorModeMachine.java (6)
LateralActiveModeMachine.java (13)
LateralApproachModeMachine.java (5)
LateralNavigationModeMachine.java (7)
LateralRollModeMachine.java (8)
VerticalActiveModeMachine.java (30)
VerticalAltitudeSelectModeMachine.java (4)
VerticalApproachModeMachine.java (5)
VerticalFlightLevelChangeModeMachine.java (7)

Figure 1: **Directory Structure Showing FGS Faults**

```
@T(Crew_Interface_Flight_Control_Panel_mon_FD_Switch_left = ON
   OR Crew_Interface_Flight_Control_Panel_mon_FD_Switch_right = ON)
WHEN (NOT Aircraft_Data_sources_term_Overspeed AND NOT
      Autopilot_Modes_Engage_term_AP_Engaged)
```

with appropriate substitution, this can be simplified to @T(A OR B) AND (not C AND not D). To generate test cases for this transition, it is expanded to:
== not(A OR B) AND (A' OR B') AND (not C AND not D)
== not A AND not B AND (A' OR B') AND (not C AND not D)
which yields the following test case requirements (not all test cases are shown):

```
    A  B  A' B' C  D
 1) f  f  t  t  f  f  // should transition to OFF
 4) f  f  t  t  t  f  // should not transition, stay in ON
 5) f  f  t  t  f  t  // should not transition, stay in ON
```

When executed, tests 4 and 5 took the transition to OFF. Analysis of the results made it seem like the code implemented (`not C OR not D`) instead of (`not C AND not D`). This was implemented in the `update()` method in `FlightDirectorModeMachine.java`, which contained the statements:

```
1) Lines beginning with '//' are comments and are ignored.
2) Blank lines are ignored.
3) The mode transition table is chosen with the line:
   Table <#>
   Where '#' refers the to the table number as given in the FGS specification.
4) The mode is set with the line:
   PreviousMode <MODENAME>
5) A mode condition table update is called with the line:
   Update
6) Monitored variables are set by lines such as:
   <VARNAME> [PC] <VALUE>
   P = Previous value
   C = Current value
   Boolean values should be F or T.
7) Previous values for term variables are set by lines such as:
   <VARNAME> <VALUE>

The update() methods are also called after all input lines have been read.
```

Figure 2: **Test Driver Input Specifications**

```
((aircraft.navigation.overspeed().previousEquals(false))  ||
 (aircraft.flightDirector.apEngageRequested().previousEquals(false)))))
```

The "`||`" (OR) should have been "&&" (AND).

The problem in the mappings tables was with the mappings of monitored variables to implementation. For the variable
`Flight_Modes_Vertical_Approach_mon_Within_Vertical_APPR_Capture_Window`,
the mapping file listed the method to set its previous value to be
`aircraft.aircraftDSO.withinVerApprCaptWindow().setCurrentValue()`.
It should have been
`aircraft.aircraftDSO.withinVerApprCaptWindow().setPreviousValue()`.

The problem with the SCR specification was a duplication in the SCR specification. In the last transition for mode transition table Flight_Modes_Flight_Director_Mode (table 33), one term appears twice:

```
OR
@(T(Crew_Interface_Throttles_term_GA_Pressed) OR

⋮

OR
@(T(Crew_Interface_Throttles_term_GA_Pressed)
```

This duplication was an artifact of the way in which the SCR specifications were generated, which was by using the Prefer tool, which is a Rockwell Collins proprietary requirements analysis tool based on the CoRE method. This had no effect on the tests.

Mode Transition	Full Predicate		T-Vec	
Table	Num Tests	Faults Found	Num Tests	Faults Found
Table 1	24	18	50	17
Table 2	20	17	20	23
Table 3	16	10	41	10
Table 4	15	16	22	16
Table 5	30	15	63	23
Table 6	44	12	2763	76
Table 7	138	29	237	34
Table 8	12	18	18	16
Table 9	12	19	18	17
Table 10	43	19	130	21
Table 11	301	53	266	57
Table 12	42	45	49	37
Table 13	18	25	30	22
Table 14	20	22	25	30
TOTAL	735	133	3732	128
Percentage		86%		83%

Table 1: Faults Found per Mode Transition Table.

3.4 Results and Analysis

All full predicate and T-Vec tests were run on all faulty versions of FGS. The data from running these tests on FGS are shown in Table 1. For each mode transition table in FGS, the table shows the total number of tests for each test technique, and the faults found by each set of tests. The mode transition table names are given in Table 2; they are the same names used in Miller and Hoech's FGS report [MH97]. The TOTAL row shows the total number of tests and the total number of faults found by each test technique. Note that the total faults found are **not** the sum of the number of faults found. The tests from the different tables found overlapping sets of faults, so the TOTAL line is not the sum of the columns. Rather, it is the number of **unique** faults found.

As can be seen, the fault detection abilities of the two techniques were very similar. Analysis of the actual faults showed that the two techniques found very close to the same set of faults. Out of 155 faults inserted into FGS, 20 were not found by either set of tests, 125 were found by both the full predicate and the T-Vec tests, and ten were only found by one set of tests (seven by the full predicate but not the T-Vec tests, and three by the T-Vec but not the full predicate tests).

Table 3 shows a different view of the data, the number of faults found in each FGS software layer. These layers are illustrated in the directory structure in Figure 1. There are several interesting things about this data. In particular, both sets of tests found 11 faults in the `CommonTypes` layer. This is somewhat surprising because `CommonTypes` contains classes that are not directly described by the functional specifications. Although the tests found a lower percentage of the `CommonTypes` faults, the fact that so many were found is encouraging support for specification-based testing. If `CommonTypes` is removed from this table, the FP tests found 89% of the faults, and the TVec tests found 86%.

Some of the 20 faults not found by either set of tests were examined by hand. Although it was not proved, it appears to be possible to detect all of them. Most of these 20 faults were not in methods that directly implemented the mode transition tables, but in auxiliary methods for condition and term tables. Recall that the tests used in the study were derived from the mode

Table Number	Table Name
Table 1	Aircraft Data Sources
Table 2	Autopilot Modes Disengaged Submode
Table 3	Autopilot Modes Engage
Table 4	Autopilot Modes Engaged Submode
Table 5	Flight Modes Flight Director Cues
Table 6	Flight Modes Flight Director Mode
Table 7	Flight Modes Lateral Active
Table 8	Flight Modes Lateral Approach
Table 9	Flight Modes Lateral Navigation
Table 10	Flight Modes Lateral Roll
Table 11	Flight Modes Vertical Active
Table 12	Flight Modes Vertical Altitude Select
Table 13	Flight Modes Vertical Approach
Table 14	Flight Modes Vertical Flight Level Change

Table 2: Mode Transition Table Names.

FGS Layer	Faults	FP Coverage	TVec Coverage	FP Percent	TVec Percent
FlightDirectorLayer	105	100	97	95%	92%
NavigationLayer	26	18	17	69%	65%
CommonTypes	19	11	11	58%	58%
ControlsLogicLayer	5	3	3	60%	60%
TOTAL	155	132	128	85%	83%

Table 3: Faults Found in Each FGS Layer.

transition tables, so they did not fully exercise all of the code that was written for the other tables.

The ten faults found by only one technique were examined by hand, but there is no obvious reason why they were found by one sets of tests but not the other. Thus, it was concluded that both test techniques were virtually identical in fault detection power, and that **both** hypotheses from Section 3.1 should be **rejected**.

The cost, however, is a different matter. An original goal of the experiment was to carefully compare both techniques in terms of cost, but this was more difficult than it might seem. First, T-Vec is entirely automated, and the full predicate criterion is not. Thus, it took several weeks of human effort to generate the full predicate tests. This comparison, however, would compare the current state of the tools, not the testing techniques, because it is entirely possible to automate FP test generation. There was also a cost with preparing the tests. Use of both techniques requires preparation on the part of the tester to create a mapping from the specification variables to the program variables. In this study, the same information was used for both techniques, so the cost could not easily be separated. For the FP tests, a general purpose test driver was generated that incorporated the variable mapping information used by T-Vec. It is not clear whether production of this driver should be counted as a cost of using FP testing, and since the mapping files produced for the T-Vec method were used to produce the driver, it is difficult to apportion the preparation cost between the two methods. It also took several hours to break the excessively large T-Vec driver into compilable size drivers. One large advantage of both methods is that the expected outputs for test cases can be produced automatically. This obviates a cost that is often a major cost of testing.

The major measurable difference in the cost of the two methods is in terms of the number of test cases. T-Vec created 3732 tests compared with only 735 FP tests. The primary difference is due to the goals of the testing techniques. As stated earlier, T-Vec attempts to satisfy MCDC structural coverage at the source code level, and may achieve MCC coverage. Full predicate testing attempts to exercise all predicates in the specification, in a manner that is very similar to MCDC. Since MCC requires 2^N tests in the number of clauses in the predicates, it is not surprising that use of T-Vec results in more tests. This almost 5 to 1 ratio means that the cost of using T-Vec is more than using full predicate tests. We can express this difference by defining an efficiency rating. The *test case efficiency* is the number of faults found per test case, computed by dividing the number of faults by the number of test cases. In this experiment, the test case efficiency for full predicate coverage was .18 and the test case efficiency for T-Vec was .03. Thus we conclude that full predicate testing is **not more effective** than T-Vec testing, but may be **more efficient**.

4 PROOF-OF-CONCEPT AUTOMATIC TEST DATA GENERATION TOOL

SpecTest is a proof-of-concept tool that generates test cases from SCR and UML specifications [OA99] according to the specification-based test criteria. The SCR and UML specifications that SpecTest can process are case tool specific. The SCR specifications are generated by the SCR* Toolset [HKL97], which was developed by the Naval Research Laboratory. The UML specifications were generated by Rational Software Corporation's Rational Rose, hereafter Rose [Cor98].

SpecTest parses SCR and Rose specification files into a consistent intermediate form. This intermediate form is then analyzed, and tests are generated. The tester can choose inputs from either type of specification file, and can select which testing criterion to satisfy.

SCR* Toolset supports the specification of the following items:

- Type Dictionary
- Mode Class Dictionary
- Constant Dictionary
- Variable Dictionary
- Specification Assertion Dictionary
- Environmental Assertion Dictionary
- Enumerated Monitored Variable Dictionary
- Controlled Variable Dictionary
- Mode Class Tables
- Term Variable Tables
- Controlled Variable Tables

The specifications are saved as ASCII text files in the above order. There is no restriction on the file name, or on the extension. The structure of the text file is shown in Figure 3.

The following assumptions were made about the SCR specification text file:

- @T, @F denote trigger events
- AND denotes logical and
- Only one mode class
- Boolean variables
- Single variable change in event
- None/Single/Multiple variables in condition
- State transitions are deterministic

Type Dictionary

`TYPE`	*Type Name*
`BASETYPE`	*Base Type Name*
`UNITS`	*Unit Name*
`COMMENT`	*Comments for the type usage*

Mode Class Dictionary

`MODECLASS`	*Mode Class Name*
`MODES`	*List of modes separated by comma*
`INITMODE`	*Initial Mode*
`COMMENT`	*Comments for the mode class usage*

Constant Dictionary

`CONSTANT`	*Constant Name*
`TYPE`	*Type Name*
`VAL`	*Value*
`COMMENT`	*Comments for the constant*

Variable Dictionary

`MON`	*Name of a monitored variable*
`TYPE`	*Type Name*
`INITVAL`	*Initial value*
`ACCURACY`	*Accuracy*
`COMMENT`	*Rules for value assignment*

`CON`	*Name of a controlled variable*
`TYPE`	*Type Name*
`INITVAL`	*Initial value*
`ACCURACY`	*Accuracy*
`COMMENT`	*Rules for value assignment*

Specification Assertion Dictionary

`ASSERTION`	*Name of an assertion*
`EXPR`	*Expression*
`COMMENT`	*Explanation of assertion*

Environmental Assertion Dictionary

Enumerated Monitored Variable Table

Event, Mode Transition, and Condition Functions

`EVENTFUNC`	*Event function table name*
`MCLASS`	*Mode class name*
`MODES`	*Mode name*
`EVENTS`	*Event1, Event2*
`ASSIGNMENTS`	*Value1, Value2*

Figure 3: **SCR Specification Text File Structure**

```
CONDFUNC     Condition function table name
CONDITIONS   Condition1, Condition2
ASSIGNMENTS  Value1, Value2

MODETRANS    Mode transition table name
FROM         State name
EVENT        Event
WHEN         List of Disjunctive Conditions
TO           State name
```

Figure 3: **SCR Specification Text File Structure – continued**

```
Logical Models
    object Class
        classAttributes

        -------------- State Transition:  Logical -------------


        State Machine
            Object State /* StartState, Normal, EndState */
                State Transition
                State Machine
                    Object State /* Normal */

        -------------- State Transition:  Physical ------------


        State Diagram
            State View /* StartState, Normal, EndState */
            Transition View

    object Association
        object Role
Logical Presentations
    object ClassDiagram /* with grouping and name */
        object ClassView
            Association View
            Role View
            Inheritance View
```

Figure 4: **Structure of MDL File for Class Diagram and State Transition Diagram**

SpecTest parses a Rose specification file (called an MDL file) to get the semantic meanings of the specifications. MDL files store specification information from different perspectives. There are two main categories of information, logical and physical. The specification itself is grouped into two packages: *use cases* and *object collaboration diagrams* are packaged into *Use Case Packages*, and *class diagrams* and *state transition diagrams* are packaged into *Logical Views*. Figure 4 shows the internal structure of the class diagram and state transition diagram in a MDL file.

SpecTest makes several assumptions about the UML specification file input:

- All transitions are triggered by change events.
- Events and conditions are expressed through boolean type class attributes.
- The specification is written strictly following the UML notation. For example, *when* denotes a change event and conditions are in solid brackets ([]). Because there is no way to check whether a specification is well-formed or consistent, this assumption cannot be checked. The OCL does not have a mechanism to enforce its syntactic rules on all parts of the UML specification. Also, Rose does not have a function to write the specification in OCL.
- State transitions are deterministic.

4.1 Architecture

Figure 5 is a UML class diagram that describes SpecTest. Classes are represented as boxes, each of which have three parts, the class name, data members that are declared in the class, and methods of the class. The main entry point (SpecTest) has four objects, (1) a UML specification parser, (2) a SCR specification parser, (3) a full predicate test case generator, and (4) a transition-pair test case generator.

`UMLSpecParser` reads a UML specification text file, parses it, and generates state transition table(s) for classes that have state machines. `SCRSpecParser` reads SCR specification text files, parses them, and generates state transition tables for mode classes. `FullPredicate` takes a state transition table as an input, generates test cases for the full predicate coverage criterion, and saves the test cases into an ASCII text file. `TransitionPair` takes a state transition table as input, generates test cases for the transition-pair coverage criterion, and saves the test cases in an ASCII text file.

4.2 Object Collaboration Diagrams

Object collaboration diagrams (OCDs) for generating full predicate coverage, transition coverage, and transition-pair coverage test cases are shown in Figures 6, 7, and 8. The OCD is illustrated for generating test cases for full predicate coverage, the other two OCDs are similar.

In Figure 6, *SpecTest* is a main program. It interacts with the user, gets the command to read a SCR or UML specification file, and invokes `SCRSpecParser` or `UMLSpecParser`, sending a specification file name as a parameter. `SCRSPecParser` or `UMLSpecParser` opens the file, parses it, generates a transition table, and returns the transition table to *SpecTest*. Next, *SpecTest* invokes `FullPredicate`, sending the state transition table as a parameter. `FullPredicate` generates test cases for full predicate coverage criteria, saves the test cases in files, and returns a message to *SpecTest* that test cases have been generated.

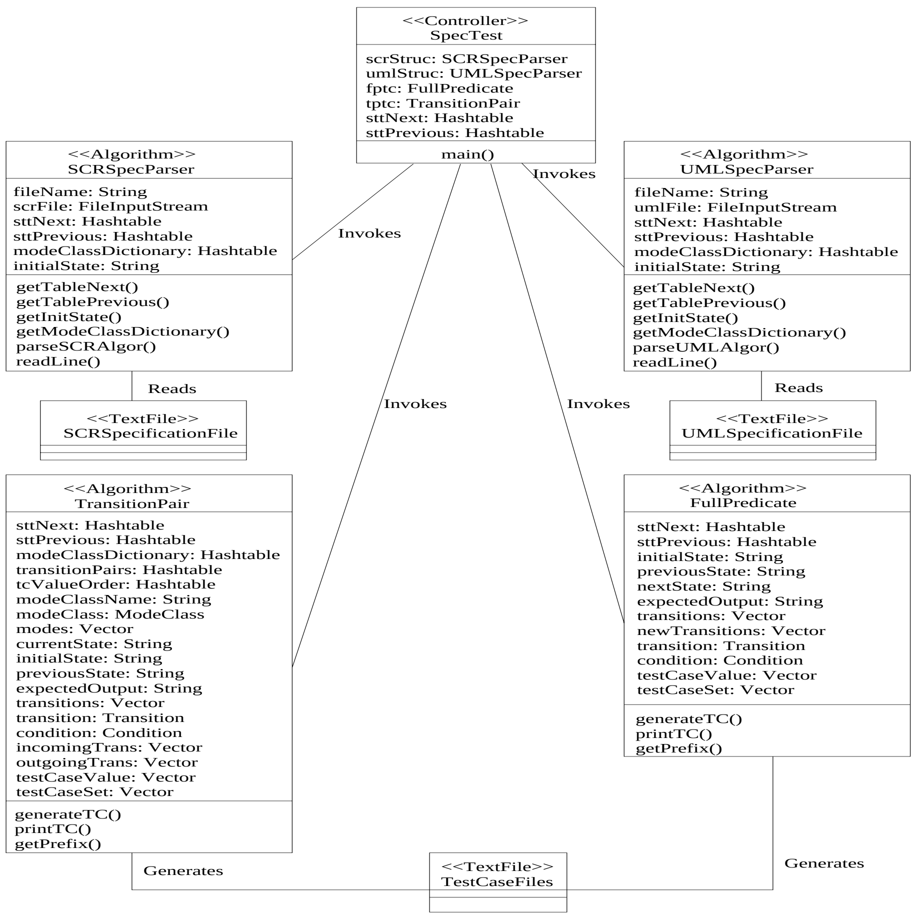


Figure 5: **Class Diagram for SpecTest Tool**

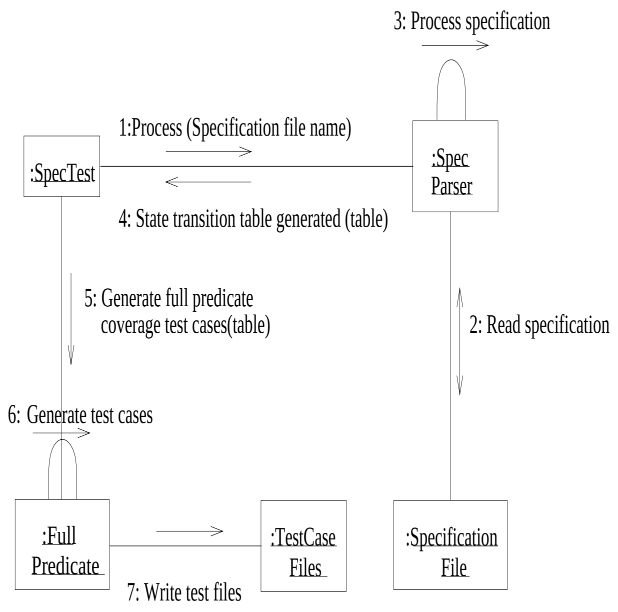


Figure 6: **OCD for Generating Full Predicate Coverage Test Cases**

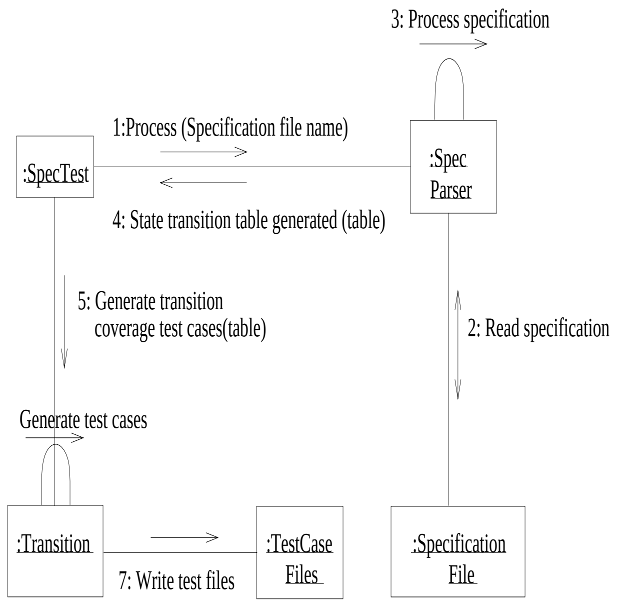


Figure 7: **OCD for Generating Transition Coverage Test Cases**

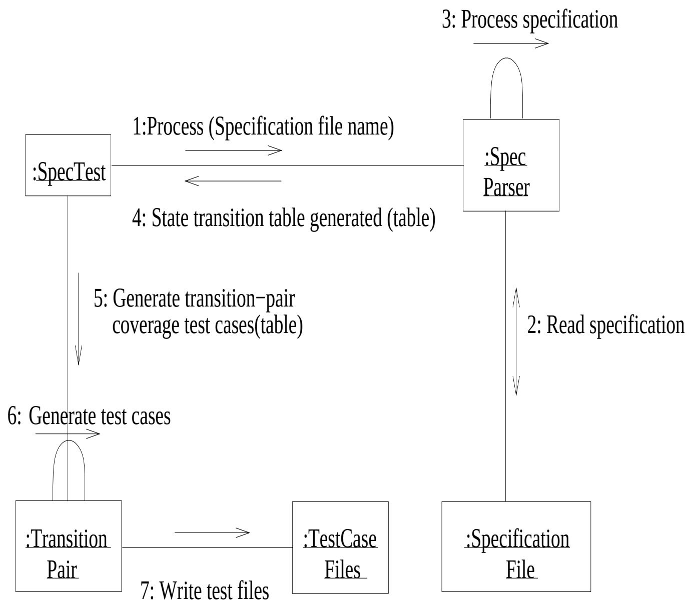


Figure 8: **OCD for Generating Transition-Pair Coverage Test Cases**

5 CONCLUSIONS

This report presents results from an empirical evaluation of the specification-based test criteria developed previously. This evaluation was done by comparing full predicate testing with tests from another specification-based testing technique, T-Vec, on faults inserted into an industrial software system. The results indicate that **neither** technique is **more effective** than the other, but that full predicate testing may be **more efficient** than T-Vec testing.

This result also presents an initial proof-of-concept tool for generating tests to satisfy the full predicate criterion. Although this tool has some significant restrictions (it only handles SCR specifications in single mode transition tables, all variables must be boolean, and the transition predicates must be single-variable expressions), the tool is integrated with the Naval Research Laboratory's SCRTool [HKL97], and shows great promise.

6 FUTURE WORK

The immediate goal of this research was to demonstrate the **practical feasibility** of the test generation criteria from the previous years. The next goals of this research are threefold: (1) to **expand the tool** to remove the restrictions so that it can be used with arbitrary SCR specifications, (2) to **extend the experimentation** to include transition-pair tests, and (3) to include a *traceability link* between the tests and the specific requirements being tested. This requirement is imposed by the FAA in DO-178B [SC-92]. The tool will allow specification-based tests to be generated more cheaply, which will allow it to be used in practical situations. It is hoped that the transition-pair tests will find faults that were not found by the full predicate and the T-Vec tests. Some encouragement for this was found by a separate smaller scale experiment that compared full predicate tests and transition-pair tests with another specification-based testing technique [AAD+]. This smaller experiment, which was carried out as a class project by a graduate student at George Mason University, found that the full predicate and transition-pair tests detected very different sets of faults.

References

[AAD+] Aynur Abdurazik, Paul Ammann, Wei Ding, , and Jeff Offutt. Evaluation of specification-based testing criteria. in preparation, 1999.

[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.

[Bla98] M. Blackburn. *Specification Transformation and Semantic Expansion to Support Automated Testing*. PhD thesis, George Mason University, Fairfax VA, 1998.

[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, Cupertina CA, 1998.

[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.

[MH97] S. P. Miller and K. F. Hoech. Specifiying the mode logic of a flight guidance system in CoRE. Technical report WP97-2011, Commercial Avionics, Rockwell Collins, Inc., Cedar Rapids, IA, 1997.

[MW94] A. P. Mathur and W. E. Wong. An empirical comparison of data flow and mutation-based test adequacy criteria. *The Journal of Software Testing, Verification, and Reliability*, 4(1):9–31, March 1994.

[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)*, Fort Collins, CO, October 1999. IEEE Computer Society Press.

[Off98] A. J. Offutt. Generating test data from requirements/specifications: Phase I final report. Technical report ISSE-TR-98-01, Department of Information and Software Systems Engineering, George Mason University, Fairfax VA, April 1998. http://www.ise.gmu.edu/techrep.

[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.

[OXL99] Jeff Offutt, Yiwei Xiong, and Shaoying Liu. Criteria for generating specification-based tests. In *Proceedings of the Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, pages 119–131, Las Vegas, NV, October 1999. IEEE Computer Society Press.

[SC-92] RTCA Committee SC-167. Software considerations in airborne systems and equipment certification, Seventh draft to Do-178A/ED-12A, July 1992.

[Won93] Weichen Eric Wong. *On Mutation and Data Flow*. PhD thesis, Purdue University, December 1993. (Also Technical Report SERC-TR-149-P, Software Engineering Research Center, Purdue University, West Lafayette, IN).

[WWH91] E. J. Weyuker, S. N. Weiss, and R. G. Hamlet. Comparison of program testing strategies. In *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification*, pages 1–10, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society Press.

Contents