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

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

Principal Investigator: A. Jefferson Offutt
George Mason University

November 3, 2000

EXECUTIVE SUMMARY

This report presents results for the Rockwell Collins Inc. sponsored project on generating test data from requirements/specifications, which started January 1, 2000. 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 give software developers the opportunity to describe exactly what services the software should provide, providing information to software designers in a clear, unambiguous manner. Formal specifications give test engineers information about the expectations of the software in a form that can be automatically manipulated.

This Phase IV, 2000 report presents progress on an empirical evaluation of the efficacy of the transition-pair criterion developed in previous years. Transition-pair tests have been developed for the Flight Guidance System (FGS) and they were run on the faulty versions of FGS developed last year. These tests and the results are summarized in this preliminary report. This report also presents progress in our test data generation tool. This tool has been significantly expanded to handle multiple SCR tables, recursively defined tables, event and condition tables, non-boolean variables, and multiple-variable expressions. It is integrated with the Naval Research Laboratory's SCRTool and Rational Software Corporation's Rational Rose tool.

Technical Report ISE-TR-01-03, Department of Information and Software Engineering, George Mason University, Fairfax VA, July 2001.

1 INTRODUCTION

There is an increasing need for effective testing of software for safety-critical applications, such as avionics, medical, and other control systems. These software systems usually have clear high level descriptions, sometimes in formal representations. Unfortunately, most system level testing techniques are only described informally. This project 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. This research project is attempting to establish formal criteria and processes for generating system-level tests from functional requirements/specifications.

In previous years, this work has resulted in a general model for developing test inputs from state-based specifications, specific criteria for defining tests, algorithms for generating and optimizing sets of test cases, empirical evaluation of the test techniques, and a preliminary proof-of-concept automatic test data generator. 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 two parallel activities: A significant expansion of the proof-of-concept test data generation tool and an empirical evaluation of the transition-pair criterion on Rockwell Collin's research version of the Flight Guidance Mode Logic System (FGS). The preliminary report also summarizes previous results (during 1997 [Off98, OXL99], 1998 [Off99] and 1999 [Off00]), and reiterates the current year goals.

2 SUMMARY OF PHASES I, II AND III

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 **criteria** for generating tests. There are four separate criteria: the complete transition sequence level, the transition-pair level, the transition level, and 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 [HL92] 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, initial tests were generated for specifications of a research version of an industrial software system supplied by Rockwell Collins, the Flight Guidance Mode Logic System (FGS). 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 [Off99]. Algorithms for test case development were developed and a small empirical evaluation of the test criteria was carried out.

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. The first was 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.

Phase III of this project was carried out during Spring and Summer 1999 [Off00]. An empirical evaluation of the full predicate specification-based testing criterion was performed, and a preliminary proof-of-concept test data generation tool was developed.

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 similar numbers 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 is designed to create full predicate and transition-pair tests from an SCR specification. During Phase III, enough of the tool was implemented to generate tests from single mode transition table SCR specifications with all boolean variables, and single-variable expression transition predicates.

2.1 Summary of Phase IV Goals

The current year research carries the previous results forward in two directions, building directly on the results from 1999 to move the specification-based testing technique developed during the first two phases of this project towards practical feasibility. The first direction is to expand the proof-of-concept tool created during Phase III to remove the restrictions so that it can be used with arbitrary SCR specifications. This includes allowing multiple tables, recursively defined tables, event and condition tables, non-boolean variables, and multiple-variable expressions. The final version of the tool now generates specification-based tests more cheaply, allowing it to be used in practical situations. The final tool was to be evaluated in two primary ways: (1) it must be able to handle the FGS SCR specification, and (2) the full predicate tests it generates are being compared with the already existing hand-generated tests from Phase III.

Second, the empirical evaluation carried out during Phase III is being extended. Specifically, the experiment is being expanded to include the transition-pair criterion. Tests are being created, the analysis of fault finding effectiveness carried out in Phase III for T-Vec tests and full predicate tests will be repeated for the transition-pair tests, and the results will be compared with the results for the previous tests. 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 [ADO00].

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 an additional conference paper are also currently in preparation.

1. Aynur Abdurazik and Jeff Offutt. Using UML Collaboration Diagrams for Static Checking and Test Generation. *Third International Conference on the Unified Modeling Language (UML '00)*, pages 383-395, York, England, October 2000.

2. Aynur Abdurazik, Paul Ammann, Wei Ding and A. Jefferson Offutt. Evaluation of Three Specification-based Testing Criteria. *The Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '00)*, pages 179–187, Tokyo, Japan, September 2000.

3. Aynur Abdurazik, Zhenyi Jin, Liz White and Jeff Offutt. Analyzing Software Architecture Descriptions to Generate System-level Tests. *Workshop on Evaluating Software Architectural Solutions (WESAS '00)*, http://www.isr.uci.edu/events/wesas2000/, Irvine, CA, May 2000.

4. 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.

5. 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.

6. 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).

7. Jeff Offutt, Generating Test Data From Requirements/Specifications: Phase III Final Report, January 2000, George Mason University Department of ISE Technical Report ISE-TR-00-02, http://www.ise.gmu.edu/techrep/.

8. 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/.

9. 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 TEST DATA GENERATION TOOL

SpecTest is a proof-of-concept tool that generates test cases from SCR specifications and UML statecharts according to the specification-based test criteria. The SCR specifications are generated by the SCR* Toolset [HKL97], which is developed by the Naval Research Laboratory. The UML statecharts are generated by Rational Software Corporation's Rational Rose, hereafter Rose [Cor98]. This report is concerned with the SCR portion of SpecTest, which is the portion that is funded by Rockwell Collins.

SpecTest parses SCR specification files into a consistent intermediate form. This intermediate form is then analyzed and tests are generated. The tester can select which testing criterion to satisfy.

SpecTest was described in detail in the 1999 final report. This report describes a major update to SpecTest. As of the end of the last year's project in August 1999, SpecTest had the following restrictions:

- SpecTest could only process mode class tables.
- The specification could only contain one table.
- Variables were restricted to be of type Boolean.
- Trigger event expressions were restricted to having one variable.

We have removed all these restrictions for full predicate test generation. SpecTest now creates full predicates tests for SCR specifications that contain all SCR tables and dictionaries (type dictionaries, mode class dictionaries, constant dictionaries, variable dictionaries, enumerated monitored variable dictionaries, controlled variable dictionaries, mode class tables, and term variable tables), and can handle an arbitrary number of tables. Two types of dictionaries, specification assertion dictionaries and environmental assertion dictionaries, are not needed for test generation and thus are not used. It also can handle SCR specifications containing variables of all numeric, enumerated, and boolean types, and expressions with multiple variables.

The complexity of the problem was significantly greater than expected, involving substantial changes to the base underlying data structures used by SpecTest.

The high level design of SpecTest is shown in Figure 1, with a UML context diagram and a high level UML class diagram in Figures 2 and 3. The main component, `SpecTest`, invokes one of the two parsers to parse either SCR specifications (`ScrSpecParser`) or UML specifications (`UMLSpecParser`). The parsers read a file created by either the SCRTool [HKL97] or Rose [Cor98], and store the information into a standard format that reflects the specification graph and the transition predicates (`Dictionaries & Specification Graph`). After the specification graph is built, either full predicate tests or transition-pair tests are created. The main component calls either `FullPred` or `TransPair` to perform this operation, and the tests are saved in the file `TestCaseFiles`. `FullPred` and `TransPair` include the algorithms from Phase II to create prefix values and to optimize tests.

The SpecTest tool is written completely in Java 1.2. Currently, the `SpecTest` component contains 20 Java classes, and uses the class package `SpecTree`, which contains another 6 classes. The primary 10 classes, which are SpecTest, SCRSpecParser, FullPred, FullPredSCR, Predicate, ConjunctElem, DisjunctElem, TermCondition, TestSetElem, PredProcessor, and the package SpecTree, which contains classes BinaryTree and BinaryNode, are shown in the partial summary class diagram in Figure 4. The class `SpecTest` contains two constructors and three methods, `ScrSpecParser` contains one constructor and 17 methods, `FullPred` contains one constructor and 11 methods, `FullPredSCR` contains one constructor and four methods, `Predicate` contains two constructors and three methods, `ConjunctElem` contains one constructor and two methods, `DisjunctElem` contains one constructor and two methods, `TermCondition` contains one constructor and four methods, `TestSetElem` contains two constructors and two methods, `PredProcessor` contains one constructor

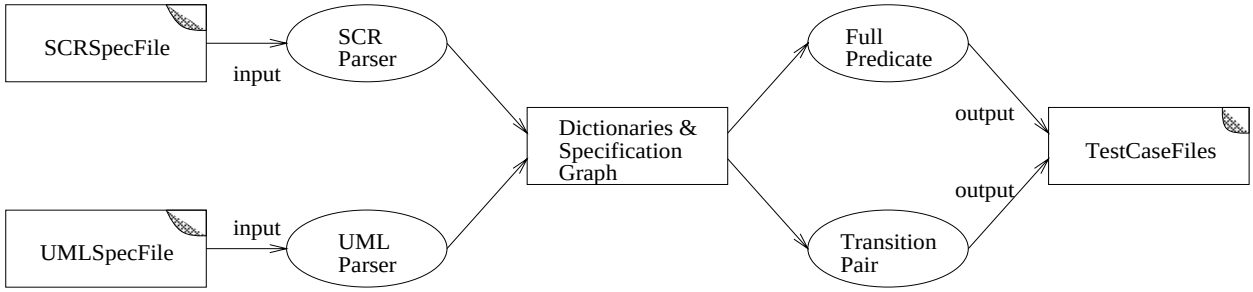

Figure 1: SpecTest **Component Design**

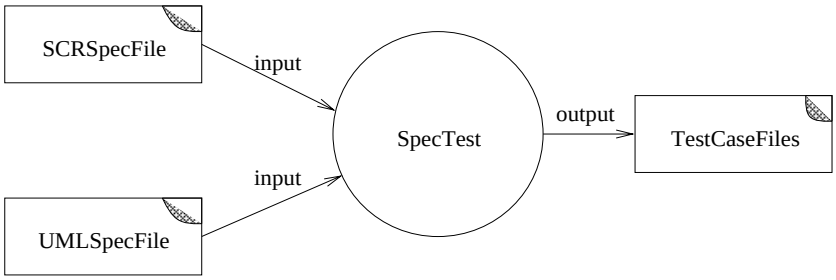

Figure 2: SpecTest **Context Diagram**

and 14 methods, `BinaryTree` contains three constructors and 16 methods, and `BinaryNode` contains three constructors and 19 methods. As can be seen, `SpecTest` can generate tests from either SCR or UML, and to satisfy either the full predicate or the transition-pair criterion. This research is concerned with testing for SCR, and the portion of the tool devoted to UML has been funded separately. This year's report is primarily concerned with updates to the full predicate component of the tool.

3.1 Assumptions

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

- @T, @F denote trigger events
- AND denotes logical and
- OR denotes logical or
- NOT denotes logical negation
- Arbitrary number of mode classes
- Boolean, Float, Integer, and Enumeration type variables
- Multiple variable change in event
- None/Single/Multiple variables in condition
- State transitions are deterministic

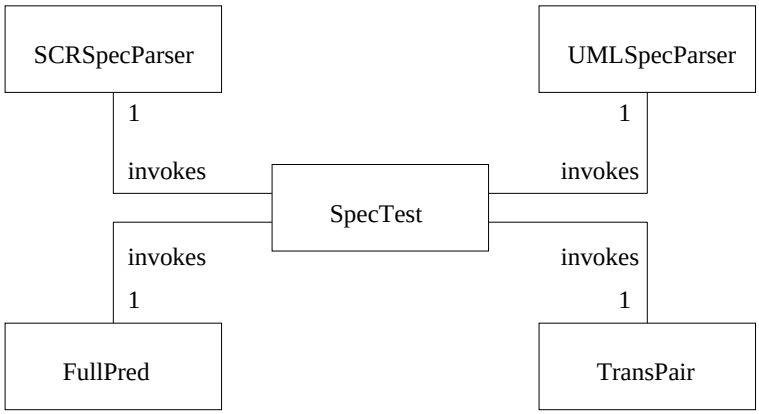

Figure 3: **High Level SpecTest Class Diagram**

3.2 Architecture

Figure 4 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. Data members and methods of classes are omitted from the diagram to allow it to fit on one page. The main entry point (SpecTest) has four main 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.

This report describes the classes that participate in parsing the SCR specification and generating tests for the full predicate criterion. There are two categories of classes, algorithmic classes and data structure classes.

1. Algorithmic Classes: There are three major tasks in generating tests from SCR specifications. First, the specification text file is scanned, then the specification is parsed and a predicate tree generated for each mode transition predicate, and finally tests are generated for each variable in the predicate tree. SpecTest parses the SCR specification according to the following grammar:

```
*************************************************************
* The Grammar of the SCR Predicates:
*
* <predicate>      -> <disjunct elem><disjunct><predicate>
* <predicate>      -> lambda
* <disjunct>       -> OR
* <disjunct>       -> lambda
* <disjunct elem>  -> <conjunct head><conjunct elem><conjunct><disjunct
* elem>
* <disjunct elem>  -> lambda
* <conjunct head>  -> NOT
* <conjunct head>  -> lambda
* <conjunct>       -> AND
* <conjunct>       -> lambda
* <conjunct elem>  -> (<predicate>)
* <conjunct elem>  -> <NBVar><ROP><NBVar>
* <conjunct elem>  -> BVarId
```

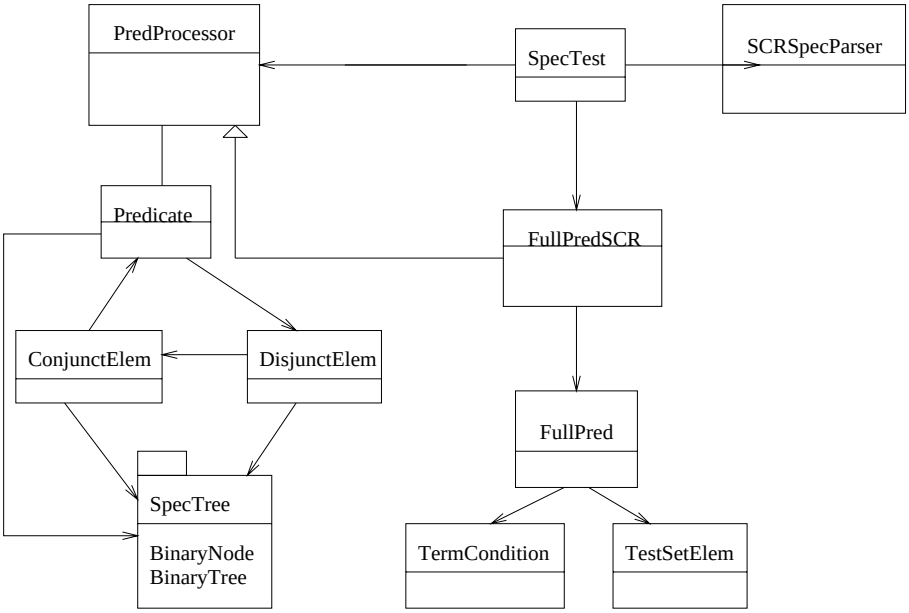

Figure 4: **Partial Summary Class Diagram for** SPECTEST **Tool**

```
* <NBVar>          -> NBVarID <==> NonBoolean(NB)MonVar, NBConVar,
*                                  NBTerm, ModeClassName
* <NBVar>          -> constant
* <NBVar>          -> (<NBVar><COP><NBVar>)
* <ROP>            -> =
* <ROP>            -> !=
* <ROP>            -> >
* <ROP>            -> >=
* <ROP>            -> <
* <ROP>            -> <=
* <COP>            -> +
* <COP>            -> -
* <COP>            -> *
* <COP>            -> /
*****************************************************************
```

The algorithmic classes category includes the following classes:

- `SCRSpecParser`: Reads in a SCR specification file, parses the specification, and generates the Mode Class Dictionary, the Variable Dictionary, the Constant Dictionary, and the Event and Condition Tables. Figures 5 through 8 shows the inner structure of the dictionaries.

  These tables show the abstract structure of each dictionary. In the implementation, each table is stored using Java `Hashtable`s. The objects of `Hashtable`s are represented by other classes. Figure 5 shows the overall view of Mode Class Dictionary structure. For each mode class, all transitions are represented twice. One is with pre and next modes, and the other is with next and pre modes. Each predicate is represented as a binary tree. It also includes the initial mode for a mode class and comments from the specification. The other tables are fairly simple in form, and resemble tables in the SCR specification.

Mode Class	Mode Class (Class)				
	modes				
	pre_mode1	predicate1	next_mode1	initMode	comment
		predicate2	next_mode2		
	next_mode1	predicate3	pre_mode1		
		predicate4	pre_mode2		
	pre_mode2	predicate5	next_mode3		
		predicate6	next_mode4		
	next_mode2	predicate7	pre_mode3		
		predicate8	pre_mode4		

Figure 5: **Data Structure of Mode Class Dictionary**

Var Name	Var Class					
	type	base type	initval	vartype	comment	accuracy

Figure 6: **Data Structure of Variable Dictionary**

Constant Name	Constant (Class)		
	type	value	comment

Figure 7: **Data Structure of Constant Dictionary**

Event (Condition) Table Name	Event (Condition) Table	
	condition	value
	condition	value
	⋮	⋮

Figure 8: **Data Structure of Event and Condition Tables**

- `PredProcessor`: Processes mode transitions in mode class tables. PredProcessor generates a predicate tree for each mode transition predicate, and saves it in the Mode Class Dictionary.
- `FullPredSCR`: Takes processed mode class dictionary and walks through all the mode transition tables. For each transition predicate, FullPredSCR invokes FullPred to generate tests for the full predicate criterion.
- `FullPred`: Takes a binary predicate tree and walks through the tree from each leaf node that is a variable to generate test cases. FullPred generates two test cases for each variable – one for the true case and one for the false case.
- `Predicate`: Generates a predicate tree from a given condition that is described with a string. As shown in the grammar, `Predicate` collaborates with `ConjunctElem` and `DisjunctElem` to generate a predicate tree for a given predicate.
- `ConjunctElem`: See `Predicate`.
- `DisjunctElem`: See `Predicate`.

2. Data Structure Classes: There are eight data structure classes. These all have a simple structure and have methods to set values, get values, and constructors.
   - `Var`
   - `Type`
   - `ECTable`
   - `Constant`
   - `Trans`
   - `ModeClass`
   - `TestSetElem`
   - `VarvaluePair`

3.3 Decisions in Algorithms

There are a number of decisions that were made for this tool that need to be documented. Some of the decisions are complicated, and were made for extremely technical reasons, and others may need to be re-visited in the future.

- Correlated variables ($A > 0$ and $A < 0$).
  When the same variable appears twice in an expression, those two variables are said to be correlated, and the test data generator must recognize that both occurrences need to have the same variable.

  This is a common problem in software testing that has been discussed as far back as Myers' early textbook [Mye79], and appears in almost every testing technique. In fact, it is related to the well known *aliasing* problem in compilers and software analysis. Although the problem is theoretically undecidable, the correlated variable problem has less impact in specification-based testing, because it happens less often and we usually have more degrees of freedom when selecting values for variables.

  Figure 9 gives an algorithm for a partial solution to the problem of correlated variables in a predicate. The inputs are a variable in the predicate, the test set that contains already chosen test values for the current variable that is being tested, the status ID of the variable, a value that the variable must be compared with, and the name of a relational operator. The output is the value assigned to the variable.

```
algorithm:        CorVar (testSet, var, ID, cmpVal, relOpName)
input:            A test set, a variable, the status ID of the
                  variable, e.g. before or after value, a value to be compared
                  and the name of a relational operator.
output:           Value to the given variable.
output criteria:  Value satisfies the predicate.
declare:          curTC: Current test case in the testSet.
                  curVarName:  Name of variable of the current
                  test case in the testSet.
                  curVarID: Status ID of variable of the current test case.
                  inc:  An incrementor to calculate the value of a variable.
                  from the given value.

CorVar (testSet, var)
BEGIN -- Algorithm CorVar
    WHILE (testSet has more test cases) LOOP
        curTC      = testSet.nextElement();
        curVarName = curTC.varName;
        curVarID   = curTC.varID;
        curVarVal  = curTC.varValue;
        IF ((curVarName equals varName) ⋀ (curVarID equals ID)) THEN
            IF ((relOpName equals ≤) ⋁ (relOpName equals <)) THEN
                varValue = cmpVal - inc;
                IF (curVarVal < varValue) THEN
                    varValue = curVarVal;
                END IF
            ELSE IF (repOpName equals =) THEN
                varValue = cmpVal;
            ELSE IF ((repOpName equals >) ⋁ (repOpName equals ≥)) THEN
                varValue = cmpVal + inc;
                IF (curVarVal > varValue) THEN
                    varValue = curVarVal;
                END IF
            ELSE
                Report error
                EXIT
            END IF
        END IF
    END LOOP
END Algorithm CorVar
```

Figure 9: **The Correlated Variable Algorithm**

- Infeasible predicates.
  Occasionally, there are predicates or combinations of predicates in the specification that cannot be satisfied. Infeasible specifications often result when term variables are substituted with corresponding conditions or events from event or condition tables.

  This is a specific instance of a more general problem, commonly called the *feasible path problem*, which says that for certain structural testing criteria some of the test requirements are infeasible in the sense that the semantics of the program imply that no test case satisfies the test requirements. Equivalent mutants, unreachable statements in path testing techniques, and infeasible DU-pairs in data flow testing are all instances of the feasible path problem. Goldberg, Wang, and Zimmerman [GWZ94] define the problem as follows: "given a description of a set of control flow paths through a procedure, *feasible path analysis (FPA)* determines if there is input data that causes execution to flow down some path in the collection". Offutt and Pan [OP97] generalized it to the *feasible test problem (FTP)*: given a requirement for a test case, the feasible test problem is to determine if there is input data that can satisfy the requirement. Infeasible predicate analysis is one instance of this problem.

  Although good partial solutions exist for this problem, we have not implemented them in our tool. Again, this problem occurs less often in specification-based testing than code-based testing, primarily because the predicates tend to be simpler in nature.

- Poorly chosen values.
  The general problem of choosing values to satisfy predicates is undecidable, thus, like any test data generator, SpecTest must occasionally make choices of values based on incomplete information. This means that SpecTest will occasionally fail to satisfy a test requirement. SpecTest does not yet have the ability to verify the consistency and correctness of values that are chosen, and does not use a search process to refine invalid choices. It is interesting to note that SpecTest did not fail even once on the FGS specifications.

3.4 Results

One of the mechanisms used to evaluate SpecTest was to use it to generate tests for Rockwell-Collins' Flight Guidance System (FGS) SCR specification. This has been successfully accomplished, and tests for the 14 mode transition tables are provided in subsections 3.4.1 through 3.4.14. There is still one outstanding anomaly. In a number of cases, the expected output and actual outputs are different. Examples are supplied below, and we are still exploring this problem.

3.4.1 Aircraft_Data_Sources - Table 7

Table Aircraft_Data_Sources has two states and four state transitions. SpecTest generated 24 full predicate test cases for this mode transition class. The actual outputs of all the test cases are the same as the expected outputs.

3.4.2 Autopilot_Modes_Disengaged_Submode - Table 12

Mode Class Autopilot_Modes_Disengaged_Submode has three states: NORMAL, WARNING, and CLEARED. There are three transitions in total. SpecTest generated 24 test cases for this mode class. The actual outputs of four test cases are not the same as the expected output. Although we have checked the predicates and the test cases, this problem is still unresolved.

An example test case that gives anomalous results is:

```
// t12tc006
```

```
// Table 12: Autopilot_Modes_Disengaged_Submode
// Previous Mode:  CLEARED
// Expected Mode:  CLEARED

// Transition Predicate: @T(Autopilot_Modes_Engage = DISENGAGED) AND
// NOT ((@T(Crew_Interface_Throttles_term_GA_Pressed) AND (NOT
// Aircraft_Data_Sources_term_Overspeed')))

// Expanded Predicate:  (Autopilot_Modes_Engage = ENGAGED AND
// (Crew_Interface_Flight_Control_Panel_mon_AP_Disconnect_Bar != DOWN)
// AND (Crew_Interface_Flight_Control_Panel_mon_AP_Disconnect_ Bar' =
// DOWN)) AND (((Crew_Interface_Throttles_term_GA_Pressed) OR
// NOT Crew_Interface_Throttles_term_GA_Pressed' OR
// (Aircraft_Data_Sources_term_Overspeed')))

Table   12
PreviousMode CLEARED
Crew_Interface_Throttles_term_GA_Pressed          F
Autopilot_Modes_Engage          ENGAGED
Crew_Interface_Flight_Control_Panel_mon_AP_Disconnect_Bar    C      DOWN
Crew_Interface_Flight_Control_Panel_mon_AP_Disconnect_Bar    P      UP
Aircraft_Data_Sources          SPEED_OK
Crew_Interface_Throttles_mon_GA_Switch_right    C      ON
Crew_Interface_Throttles_mon_GA_Switch_left    C      ON
```

The expanded predicate is derived from the given transition predicate, then the test case is generated for the expanded predicate. If the expanded predicate is evaluated with the above test case, the predicate evaluates to FALSE, thus the actual output should be CLEARED. However, when the test case is on the implementation, the actual output is WARNING.

3.4.3 Autopilot_Modes_Engage

Mode Class Autopilot_Modes_Engage has two states and four transitions. SpecTest generated 20 full predicate test cases for this mode class, and all of them have the same actual and expected outputs.

3.4.4 Autopilot_Modes_Engaged_Submode - Table 17

Mode Class Autopilot_Modes_Engage_Submode has three states and four transitions. SpecTest generated 18 full predicate test cases for this mode class, and all of them have the same actual and expected outputs.

3.4.5 Flight_Modes_Flight_Director_Cues - Table 31

Mode Class Flight_Modes_Flight_Director_Cues has three states and four transitions. SpecTest generated 52 full predicate test cases for this mode class. Four of the test cases have different actual and expected outputs.

Following is one of the test cases that demonstrates the problem.

```
// t31tc050
// Table 31: Flight_Modes_Flight_Director_Cues
// Previous Mode:  Cues
```

```
// Expected Mode:  Cues

// Transition Predicate: @F(Flight_Modes_Flight_Director_Mode = ON)

// Expanded Predicate:  (Flight_Modes_Flight_Director_Mode = ON AND
// (Crew_Interface_Flight_Control_Panel_mon_FD_Switch_left != ON AND
// Crew_Interface_Flight_Control_Panel_mon_FD_Switch_right !  = ON) AND
// (Crew_Interface_Flight_Control_Panel_mon_FD_Switch_left' = ON OR
// Crew_Interface_Flight_Control_Panel_mon_FD_Switch_right' = ON) AND
// (NOT Aircraft_Data_Sources_term_Overspeed AND
//  NOT  Autopilot_Modes_Engage_term_AP_Engaged))

Table   31
PreviousMode Cues
Autopilot_Modes_Engage_term_AP_Engaged          T
Flight_Modes_Flight_Director_Mode          ON
Aircraft_Data_Sources_term_Overspeed          F
Crew_Interface_Flight_Control_Panel_mon_FD_Switch_right    C      ON
Crew_Interface_Flight_Control_Panel_mon_FD_Switch_left    C      ON
Crew_Interface_Flight_Control_Panel_mon_FD_Switch_right    P      OFF
Crew_Interface_Flight_Control_Panel_mon_FD_Switch_left    P      OFF
```

3.4.6 Flight_Modes_Flight_Director_Mode - Table 33

Mode Class Flight_Modes_Flight_Director_Mode has two states and five transitions. SpecTest generated 52 full predicate test cases for this mode class. Four of test cases have different actual and expected outputs.

Following is one of the test cases that exhibits the problem.

```
// t33tc051
// Table 33: Flight_Modes_Flight_Director_Mode
// Previous Mode:  OFF
// Expected Mode:  ON

// Transition Predicate: @T(Crew_Interface_Flight_Control_Panel_mon_VS_Switch = ON)
// OR @T(Crew_Interface_Flight_Control_Panel_mon_ALT_Switch = ON) OR
// @T(Crew_Interface_Flight_Control_Panel_mon_FLC_Switch = ON) OR
// @T(Crew_Interface_Throttles_term_GA_Pressed) OR
// @T(Crew_Interface_Flight_Control_Panel_mon_HDG_Switch = ON) OR
// @T(Crew_Interface_Flight_Control_Panel_mon_NAV_Switch = ON) OR
// @T(Crew_Interface_Flight_Control_Panel_mon_APPR_Switch = ON) OR
// @T(Crew_Interface_Throttles_term_GA_Pressed)

// Expanded Predicate:  (Crew_Interface_Flight_Control_Panel_mon_VS_Switch != ON)
// AND (Crew_Interface_Flight_Control_Panel_mon_VS_Switch' = ON) OR
// (Crew_Interface_Flight_Control_Panel_mon_ALT_Switch != ON) AND
// (Crew_Interface_Flight_Control_Panel_mon_ALT_Switch' = ON) OR
// (Crew_Interface_Flight_Control_Panel_mon_FLC_Switch != ON) AND
// (Crew_Interface_Flight_Control_Panel_mon_FLC_Switch' = ON) OR
// (NOT  Crew_Interface_Throttles_term_GA_Pressed) AND
// Crew_Interface_Throttles_term_GA_Pressed' OR
```

```
// (Crew_Interface_Flight_Control_Panel_mon_HDG_Switch != ON) AND
// (Crew_Interface_Flight_Control_Panel_mon_HDG_Switch' = ON) OR
// (Crew_Interface_Flight_Control_Panel_mon_NAV_Switch != ON) AND
// (Crew_Interface_Flight_Control_Panel_mon_NAV_Switch' = ON) OR
// (Crew_Interface_Flight_Control_Panel_mon_APPR_Switch != ON) AND
// (Crew_Interface_Flight_Control_Panel_mon_APPR_Switch' = ON) OR
// (NOT Crew_Interface_Throttles_term_GA_Pressed) AND
// Crew_Interface_Throttles_term_GA_Pressed'

Table   33
PreviousMode OFF
Crew_Interface_Throttles_term_GA_Pressed          F
Crew_Interface_Flight_Control_Panel_mon_APPR_Switch    C      OFF
Crew_Interface_Flight_Control_Panel_mon_APPR_Switch    P      ON
Crew_Interface_Flight_Control_Panel_mon_NAV_Switch    C      OFF
Crew_Interface_Flight_Control_Panel_mon_NAV_Switch    P      ON
Crew_Interface_Flight_Control_Panel_mon_HDG_Switch    C      OFF
Crew_Interface_Flight_Control_Panel_mon_HDG_Switch    P      ON
Crew_Interface_Throttles_term_GA_Pressed          T
Crew_Interface_Flight_Control_Panel_mon_FLC_Switch    C      OFF
Crew_Interface_Flight_Control_Panel_mon_FLC_Switch    P      ON
Crew_Interface_Flight_Control_Panel_mon_ALT_Switch    C      OFF
Crew_Interface_Flight_Control_Panel_mon_ALT_Switch    P      ON
Crew_Interface_Flight_Control_Panel_mon_VS_Switch    C      OFF
Crew_Interface_Flight_Control_Panel_mon_VS_Switch    P      ON
Crew_Interface_Throttles_mon_GA_Switch_left    C      ON
Crew_Interface_Throttles_mon_GA_Switch_right    C      OFF
Crew_Interface_Throttles_mon_GA_Switch_right    C      OFF
Crew_Interface_Throttles_mon_GA_Switch_left    C      OFF
```

3.4.7 Flight_Modes_Lateral_Active - Table 34

Mode Class Flight_Modes_Lateral_Active has six states and 14 transitions. SpecTest generated 194 full predicate test cases for this mode class. 31 test cases have different actual and expected outputs.

Following is one of the test cases that exhibits the problem.

```
// t34tc125
// Table 34: Flight_Modes_Lateral_Active
// Previous Mode:  NAV
// Expected Mode:  ROLL

// Transition Predicate:
// @C(Navigation_Sources_mon_Selected_Nav_Source) OR
// (@C(Navigation_Sources _term_Nav_Source_Frequency)  AND
// ((Navigation_Sources_term_Selected_Nav_Type = VOR) OR
// (Navigation_Sources_term_Selected_Nav_Type = LOC)))

// Expanded Predicate:  Navigation_Sources_mon_Selected_Nav_Source !=
// Navigation_Sources_mon_Selected_Nav_Source' OR
// (Navigation_Sources_term_Nav_Source_Frequency !=
```

```
// Navigation_Sources_term_Nav_Source_Frequency' AND
// ((Navigation_Sources_term_Selected_Nav_Type = VOR) OR
// (Navigation_Sources_term_Selected_Nav_Type = LOC)))

Table   34
PreviousMode NAV
Navigation_Sources_term_Nav_Source_Frequency          108.0
Navigation_Sources_term_Nav_Source_Frequency'          109.0
Navigation_Sources_term_Selected_Nav_Type          LOC
Navigation_Sources_term_Selected_Nav_Type          VOR
Navigation_Sources_mon_Selected_Nav_Source    P      FMS_1
Navigation_Sources_mon_Selected_Nav_Source    C      FMS_1
```

3.4.8 Flight_Modes_Lateral_Approach - Table 40

Mode Class Flight_Modes_Lateral_Approach has three states and three transitions. SpecTest generated 22 full predicate test cases for this mode class. Seven of test cases have different actual and expected outputs.

Following is one of the test cases that exhibits the problem.

```
// t40tc007
// Table 40: Flight_Modes_Lateral_Approach
// Previous Mode:  Armed
// Expected Mode:  CLEARED

// Transition Predicate: @F(Flight_Modes_Lateral_Active = APPR)

// Expanded Predicate:  (Flight_Modes_Lateral_Active = APPR AND
// Navigation_Sources_mon_Selected_Nav_Source !=
// Navigation_Sources_mon_Selected_Nav_Source' OR
// (Navigation_Sources_term_Nav_Source_Frequency !=
// Navigation_Sources_term_Nav_Source_Frequency' AND
// ((Navigation_Sources_term_Selected_Nav_Type = VOR) OR
// (Navigation_Sources_term_Selected_Nav_Type = LOC))))

Table   40
PreviousMode Armed
Navigation_Sources_term_Nav_Source_Frequency          108.0
Navigation_Sources_term_Nav_Source_Frequency'          109.0
Flight_Modes_Lateral_Active          APPR
Navigation_Sources_term_Selected_Nav_Type          LOC
Navigation_Sources_term_Selected_Nav_Type          VOR
Navigation_Sources_mon_Selected_Nav_Source    P      FMS_1
Navigation_Sources_mon_Selected_Nav_Source    C      FMS_1
```

In this example, the expected output is CLEARED, but the actual output was ARMED.

3.4.9 Flight_Modes_Lateral_Navigation - Table 41

Mode Class Flight_Modes_Lateral_Navigation has three states and three transitions. SpecTest generated 10 full predicate test cases for this mode class. One of the test cases have different actual and expected outputs.

Following is the test case that exhibits the problem.

```
// t41tc005
// Table 41: Flight_Modes_Lateral_Navigation
// Previous Mode:  Armed
// Expected Mode:  Track

// Transition Predicate:
// @T(Flight_Modes_Lateral_Active_term_Lateral_NAV_Track_Cond_Met)

// Expanded Predicate:
// (NOT Flight_Modes_Lateral_Active_term_Lateral_NAV_Track_Cond_Met)
// AND Flight_Modes_Lateral_Active_term_Lateral_NAV_Track_Cond_Met'

Table   41
PreviousMode Armed
Flight_Modes_Lateral_Active_term_Lateral_NAV_Track_Cond_Met          F
Navigation_Sources_mon_Selected_Nav_Source_Status    C      Valid
Flight_Modes_Lateral_Active_term_Within_Lateral_NAV_Capture_Window          T
```

The expected output is Track, but the actual output is Armed.

3.4.10 Flight_Modes_Lateral_Roll - Table 42

Mode Class Flight_Modes_Lateral_Roll has three states and eight transitions. SpecTest generated 54 full predicate test cases for this mode class. 10 test cases have different actual and expected outputs.

Following is one of the test cases that exhibits the problem.

```
// t42tc024
// Table 42: Flight_Modes_Lateral_Roll
// Previous Mode:  CLEARED
// Expected Mode:  CLEARED

// Transition Predicate: @T(Flight_Modes_Lateral_Active = ROLL)  AND
// (NOT Flight_Modes_Lateral_Active_term_Roll_LE_Threshold AND
// NOT Aircraft_Data_Sources_mon_On_Ground)

// Expanded Predicate:  (Flight_Modes_Lateral_Active = APPR AND
// Navigation_Sources_mon_Selected_Nav_Source !=
// Navigation_Sources_mon_Selected_Nav_Source' OR
// (Navigation_Sources_term_Nav_Source_Frequency !=
// Navigation_Sources_term_Nav_Source_Frequency' AND
// ((Navigation_Sources_term_Selected_Nav_Type = VOR) OR
// (Navigation_Sources_term_Selected_Nav_Type = LOC)))) AND
// (NOT  Flight_Modes_Lateral_Active_term_Roll_LE_Threshold AND
// NOT Aircraft_Data_Sources_mon_On_Ground)

Table   42
PreviousMode CLEARED
Aircraft_Data_Sources_mon_On_Ground    P      T
Flight_Modes_Lateral_Active          APPR
```

```
Flight_Modes_Lateral_Active_term_Roll_LE_Threshold          F
Navigation_Sources_term_Selected_Nav_Type          LOC
Navigation_Sources_term_Selected_Nav_Type          VOR
Navigation_Sources_term_Nav_Source_Frequency          108.0
Navigation_Sources_term_Nav_Source_Frequency'          109.0
Navigation_Sources_mon_Selected_Nav_Source    P      FMS_1
Navigation_Sources_mon_Selected_Nav_Source    C       FMS_2
```

The expected output is CLEARED, but the actual output is HDG_HOLD.

3.4.11 Flight_Modes_Vertical_Active - Table 44

Mode Class Flight_Modes_Vertical_Active has nine states: (PITCH, ALTHOLD, VS, FLC, GA, APPR, CLEARED, ALTSEL_CAPTURE, ALTSEL_TRACK) and 25 transitions. SpecTest generated 348 full predicate test cases for this mode class. 64 test cases have different actual and expected outputs.

Following is one of the test cases that exhibits problem.

```
// t44tc348
// Table 44: Flight_Modes_Vertical_Active
// Previous Mode:  PITCH
// Expected Mode:  PITCH

// Transition Predicate: @T(Crew_Interface_Flight_Control_Panel_mon_VS_Switch = ON)
// AND (Aircraft_Data_Sources_term_Overspeed')

// Expanded Predicate:  (Crew_Interface_Flight_Control_Panel_mon_VS_Switch != ON) AND
// (Crew_Interface_Flight_Control_Panel_mon_VS_Switch' = ON)
// AND Aircraft_Data_Sources_term_Overspeed'

Table   44
PreviousMode PITCH
Crew_Interface_Flight_Control_Panel_mon_VS_Switch    C      OFF
Crew_Interface_Flight_Control_Panel_mon_VS_Switch    P      OFF
Aircraft_Data_Sources          TOO_FAST
```

The expected output is PITCH, but the actual output is FLC.

3.4.12 Flight_Modes_Vertical_Altitude_Select - Table 45

Mode Class Flight_Modes_Vertical_Altitude_Select has three states (CLEARED, ACTIVE, ARMED) and four transitions. SpecTest generated 64 full predicate test cases for this mode class. 27 test cases have different actual and expected outputs.

Following is one of the test cases that exhibits the problem.

```
// t45tc003
// Table 45: Flight_Modes_Vertical_Altitude_Select
// Previous Mode:  CLEARED
// Expected Mode:  ARMED

// Transition Predicate: @F((Flight_Modes_Vertical_Active = APPR) OR
// (Flight_Modes_Vertical_Active = GA) OR
```

```
// (Flight_Modes_Vertical_Active = ALTHOLD) OR
// (Flight_Modes_Vertical_Active = CLEARED))

// Expanded Predicate:  (Flight_Modes_Vertical_Active = APPR AND
// (Flight_Modes_Lateral_Approach = Track AND (Flight_Modes_Lateral_Active = APPR
// AND
// Navigation_Sources_mon_Selected_Nav_Source
//  != Navigation_Sources_mon_Selected_Nav_Source' OR
// (Navigation_Sources_term_Nav_Source_Frequency !=
// Navigation_Sources_term_Nav_Source_Frequency' AND
// ((Navigation_Sources_term_Selected_Nav_ Type = VOR) OR
// (Navigation_Sources_term_Selected_Nav_Type = LOC))))) AND
// (((Crew_Interface_Throttles_term_GA_Pressed) OR  NOT
// Crew_Interface_Throttles_term_GA_Pressed')) AND
// (NOT Aircraft_Data_Sources_term_Overspeed'))

Table   45
PreviousMode CLEARED
Navigation_Sources_term_Nav_Source_Frequency          108.0
Navigation_Sources_term_Nav_Source_Frequency'          109.0
Flight_Modes_Vertical_Active          APPR
Flight_Modes_Lateral_Approach          Track
Flight_Modes_Lateral_Active          APPR
Navigation_Sources_term_Selected_Nav_Type          LOC
Navigation_Sources_term_Selected_Nav_Type          VOR
Navigation_Sources_mon_Selected_Nav_Source    P      FMS_1
Navigation_Sources_mon_Selected_Nav_Source    C      FMS_1
Crew_Interface_Throttles_term_GA_Pressed          T
Aircraft_Data_Sources          SPEED_OK
Crew_Interface_Throttles_mon_GA_Switch_right    C      OFF
Crew_Interface_Throttles_mon_GA_Switch_left    C      OFF
```

The expected output is ARMED, but the actual output is CLEARED.

3.4.13 Flight_Modes_Vertical_Approach - Table 47

Mode Class Flight_Modes_Vertical_Approach has three states (CLEARED, ARMED, TRACK) and 3 transitions. SpecTest generated 44 full predicate test cases for this mode class. 12 test cases have different actual and expected outputs.

Following is one of the test cases that exhibits the problem.

```
// t47tc044
// Table 47: Flight_Modes_Vertical_Approach
// Previous Mode:  TRACK
// Expected Mode:  TRACK

// Transition Predicate: @F(Flight_Modes_Lateral_Approach = Track) OR
// @T(Flight_Modes_Vertical_Active = CLEARED)

// Expanded Predicate:  (Flight_Modes_Lateral_Approach = Track AND
// (Flight_Modes_Lateral_Active = APPR AND
// Navigation_Sources_mon_Selected_Nav_Source !=
```

```
//    Navigation_Sources_mon_Selected_Nav_Source' OR
// (Navigation_Sources_term_Nav_Source_Frequency !=
//    Navigation_Sources_term_Nav_Source_Frequency' AND
// ((Navigation_Sources_term_Selected_Nav_Type = VOR) OR
// (Navigation_Sources_term_Selected_Nav_Type = LOC))))) OR
// (Flight_Modes_Vertical_Active = PITCH AND
// (Flight_Modes_Flight_Director_Mode = ON AND
// (Crew_Interface_Flight_Control_Panel_mon_FD_Switch_left != ON AND
// Crew_Interface_Flight_Control_Panel_mon_FD_Switch_right != ON) AND
// (Crew_Interface_Flight_Control_Panel_mon_FD_Switch_left' = ON OR
// Crew_Interface_Flight_Control_Panel_mon_FD_Switch_right' = ON) AND
// (NOT  Aircraft_Data_Sources_term_Overspeed AND
// NOT Autopilot_Modes_Engage_term_AP_Engaged)))

Table   47
PreviousMode TRACK
Autopilot_Modes_Engage_term_AP_Engaged          T
Flight_Modes_Lateral_Approach          Track
Flight_Modes_Lateral_Active          APPR
Flight_Modes_Vertical_Active          PITCH
Flight_Modes_Flight_Director_Mode          ON
Aircraft_Data_Sources_term_Overspeed          F
Crew_Interface_Flight_Control_Panel_mon_FD_Switch_right    C      ON
Crew_Interface_Flight_Control_Panel_mon_FD_Switch_left    C      ON
Crew_Interface_Flight_Control_Panel_mon_FD_Switch_right    P      OFF
Crew_Interface_Flight_Control_Panel_mon_FD_Switch_left    P      OFF
Navigation_Sources_term_Selected_Nav_Type          FMS
Navigation_Sources_term_Selected_Nav_Type          FMS
Navigation_Sources_term_Nav_Source_Frequency          108.0
Navigation_Sources_term_Nav_Source_Frequency'          108.0
Navigation_Sources_mon_Selected_Nav_Source    P      FMS_1
Navigation_Sources_mon_Selected_Nav_Source    C      FMS_1
```

The expected output is TRACK, but the actual output is CLEARED.

3.4.14 Flight_Modes_Vertical_Flight_Level_Change - Table 50

Mode Class Flight_Modes_Vertical_Flight_Level_Change has three states (Track, Overspeed, CLEARED) and five transitions. SpecTest generated 32 full predicate test cases for this mode class. Five test cases have different actual and expected outputs.

Following is one of the test cases that exhibits the problem.

```
// t50tc030
// Table 50: Flight_Modes_Vertical_Flight_Level_Change
// Previous Mode:  Track
// Expected Mode:  Track

// Transition Predicate: @F(Flight_Modes_Vertical_Active = FLC)

// Expanded Predicate:  (Flight_Modes_Vertical_Active = FLC AND
// (Flight_Modes_Flight_Director_Mode = ON AND
// (Crew_Interface_Flight_Control_Panel_mon_FD_Switch_left != ON AND
```

```
// Crew_Interface_Flight_Control_Panel_mon_FD_Switch_right != ON) AND
// (Crew_Interface_Flight_Control_Panel_mon_FD_Switch_left' = ON OR
// Crew_Interface_Flight_Control_Panel_mon_FD_Switch_right' = ON) AND
// (NOT Aircraft_Data_Sources_term_Overspeed AND
// NOT  Autopilot_Modes_Engage_term_AP_Engaged)))

Table   50
PreviousMode Track
Autopilot_Modes_Engage_term_AP_Engaged          T
Flight_Modes_Vertical_Active          FLC
Flight_Modes_Flight_Director_Mode          ON
Aircraft_Data_Sources_term_Overspeed          F
Crew_Interface_Flight_Control_Panel_mon_FD_Switch_right    C      ON
Crew_Interface_Flight_Control_Panel_mon_FD_Switch_left    C      ON
Crew_Interface_Flight_Control_Panel_mon_FD_Switch_right    P      OFF
Crew_Interface_Flight_Control_Panel_mon_FD_Switch_left    P      OFF
```

The expected output is Track, but the actual output is CLEARED.

4 EMPIRICAL EVALUATION

The current year's project has extending the empirical evaluation from the previous year to include the **transition-pair** criterion. During Phase III, a comparative study on a large industrial system was carried out with the goal of assessing the practical usefulness of the specification-based testing criteria. The subject program was a research version of the Flight Guidance Mode Logic System (FGS) provided by Rockwell Collins. A comparison was made between full predicate tests and tests created by the T-Vec tool [BB96, BB97] (provided by Rockwell Collins).

The test techniques were compared on the basis of fault detection ability and cost. Fault detection was measured by placing 155 faults into FGS. The faults were hand-inserted by creating a separate Java source file for each fault. To reduce the possibility of bias, faults and tests were created independently; the faults were inserted by Offutt and the tests were created by a graduate student supported on this project.

We have extended this experiment to include transition-pair tests in addition to full predicate and T-Vec tests. Encouragement for this effort comes from a separate experiment carried out in collaboration with another faculty member at George Mason University, Paul Ammann, and two graduate students [ADO00]. Three specification-based testing criteria were compared using Mathur and Wong's ProbSubsumes measure [MW94]. This measure is a "cross scoring", where tests generated for each criterion are measured against the other. The three criteria are Ammann and Black's specification-mutation coverage [ABM98], full predicate coverage, and transition-pair coverage. Tests were generated for the common Cruise control example. A novel aspect of the work is that each criterion was encoded in a model checker, and the model checker was used first to generate test sets for each criterion and then to evaluate test sets against alternate criteria. The results from this study are summarized in Table 1. As can be seen, neither the full predicate tests nor the specification mutation tests had high transition-pair scores, and the transition-pair tests did not have high full predicate or specification mutation scores. Thus, it can be inferred that transition-pair tests offer something different from full predicate and specification mutation tests.

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

Table 1: Coverage Scores of Test Criteria.

We have generated 1102 tests to satisfy the transition-pair criterion. Again, to avoid bias in the experiment, these tests were generated independently from the faults and full predicate tests by a different graduate student. These 1102 tests compare to 735 full predicate tests and 3732 T-Vec tests. Table 2 summarizes the number of test requirements and test cases generated for each mode transition table.

4.1 Transition-pair Tests

The 1102 transition-pair tests were generated by hand according to the following process:

1. Develop a specification graph for each mode
2. Identify transition-pairs from the specification graph
3. Generate test case requirements for each transition pair according to the predicates on the transitions

Table Number	Table Name	# Test Requirements	# Test Cases
Table 1	Aircraft Data Sources	8	8
Table 2	Autopilot Modes Disengaged Submode	8	8
Table 3	Autopilot Modes Engage	8	8
Table 4	Autopilot Modes Engaged Submode	7	8
Table 5	Flight Modes Flight Director Cues	24	25
Table 6	Flight Modes Flight Director Mode	23	44
Table 7	Flight Modes Lateral Active	73	210
Table 8	Flight Modes Lateral Approach	5	5
Table 9	Flight Modes Lateral Navigation	5	5
Table 10	Flight Modes Lateral Roll	25	26
Table 11	Flight Modes Vertical Active	224	701
Table 12	Flight Modes Vertical Altitude Select	25	25
Table 13	Flight Modes Vertical Approach	9	9
Table 14	Flight Modes Vertical Flight Level Change	12	12

Table 2: Transition Test Summary for FGS.

4. Generate test cases based on the test requirements

The specification graphs, transition-pairs, and test requirements are included in a separate companion document, "Transition-pair Tests for FGS".

All but 90 of the tests were executed successfully on FGS and the expected outputs were generated. For the remaining 90 tests, there are discrepancies between the actual and expected outputs. Each of these tests exhibit the same behavior. Assume the test is designed to cause transitions from state `S1` to `S2` and then to `S3` (`S1` $\rightarrow$ `S2` $\rightarrow$ `S3`). If each component of the tests is run independently, the outputs are as expected. That is, if the first part of the test is run, the correct transition from `S1` to `S2` is taken. Then, if the second part of the test is run separately, the correct transition from `S2` to `S3` is taken. But when the two test components are run together, state `S3` is not reached.

Analysis of the problem revealed a problem with the design of the test driver from last year's effort. Specifically, the test driver calls both `setPreviousValue` and `setCurrentValue` for the first inputs and then only `setCurrentValue` for subsequent inputs. Whereas this worked for full predicate tests, where only one transition was being tested, this method fails when transition-pair testing requires two subsequent transitions to be taken. Modifications to the driver are ongoing, however they could not be made in time to support this experimentation, so the 90 tests in question were removed from the experiment, leaving 1012.

4.2 Faults Inserted Into FGS

We used the same faults that were used in the 1999 experiment [Off00]. The faults were inserted by hand. The strategy was to insert faults that are (1) similar to naturally occurring faults, and (2) not trivial to detect. A total of 155 faults were created, all of which passed the Java compiler (Sun JDK 1.2.10).

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

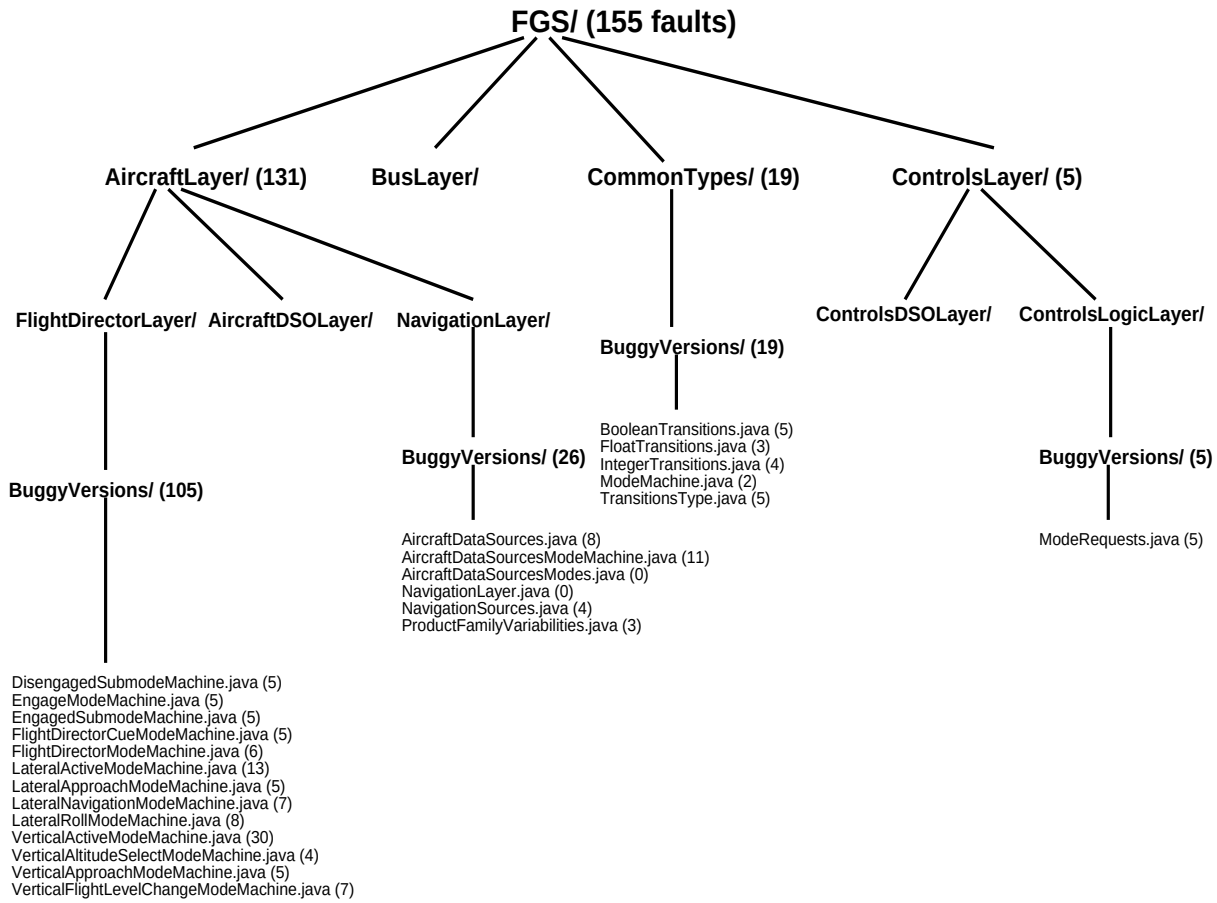

Figure 10: **Directory Structure Showing FGS Faults.**

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 (taking over 30 hours to run all 1012 TP tests on all faulty versions of FGS).

Figure 10 shows the directory structure of FGS and location of the faulty Java classes. All faulty classes are stored in subdirectories called `BuggyVersions/`. In Figure 10, 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 to be executed. 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 are able to test code that is not directly described by the specifications.

4.3 Results and Analysis

All transition-pair tests were run on all faulty versions of FGS. The data from running these tests on FGS are shown in Table 3. For each mode transition table in FGS, the table shows the total

Mode Transition Table	Full Predicate		T-Vec		**Transition-pair**	
	# Tests	Faults Found	# Tests	Faults Found	**# Tests**	**Faults Found**
Table 1	24	18	50	17	**8**	**15**
Table 2	20	17	20	23	**8**	**12**
Table 3	16	10	41	10	**8**	**7**
Table 4	15	16	22	16	**7**	**14**
Table 5	30	15	63	23	**24**	**10**
Table 6	44	12	2763	76	**23**	**8**
Table 7	138	29	237	34	**47**	**19**
Table 8	12	18	18	16	4	**6**
Table 9	12	19	18	17	**5**	**13**
Table 10	43	19	130	21	**22**	**16**
Table 11	301	53	266	57	**165**	**42**
Table 12	42	45	49	37	**24**	**29**
Table 13	18	25	30	22	**9**	**13**
Table 14	20	22	25	30	**12**	**21**
TOTAL	735	133	3732	128	**1012**	**117**
Percentage		86%		83%		**75%**

Table 3: Faults Found per Mode Transition Table.

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 4; 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 does not contain the sums of the columns. Rather, it is the number of **unique** faults found.

As mentioned before, a prior experiment [ADO00] found that transition-pair tests found a substantially different set of faults from full predicate tests on a much smaller program. To check whether this result held true for FGS, the individual faults found by each technique was analyzed. The data is shown in Table 5. In the table, the columns headed by FP, TV, and TP indicate that the fault was found by full predicate, T-Vec, and transition-pair tests, respectively. SUM indicates how many of the techniques detected the fault, and ZER, ONE, TWO, and THR separates these out for convenience. F# is the number of the fault; the numbers are not continuous because 43 faults were removed from consideration.

As can be seen, the results were quite different from our previous experiment. Only one fault (# 196) was found exclusively by transition-pair tests. Four faults were found exclusively by full predicate tests, and two by the T-Vec tests. The differences are not significant, and provides no evidence for supporting the hypothesis that transition-pair tests somehow test something different from the other two techniques. We have no explanation for why this data differs from that of the previous experiment.

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 4: Mode Transition Table Names.

5 CONCLUSIONS

This report presents significant enhancements to a tool for automatically generating test cases from requirements/specifications. This tool is now sufficient to generate tests based on all features of SCR tables.

This report also presents results from an expanded empirical evaluation of the specification-based testing criteria. Specifically, 1012 transition-pair tests have been generated for FGS and evaluated on their fault detection ability using 155 faulty versions of FGS. The results were not very positive. The transition-pair tests were only able to detect one fault that the full predicate tests did not, and the full predicate tests found 17 faults that the transition-pair tests did not. Thus, with some reluctance, it is concluded that the transition-pair technique cannot be recommended as a viable alternative to the full predicate testing technique.

F#	FP	TV	TP	Sum	ZER	ONE	TWO	THR	
	134	129	120	383	60	7	16	115	
31	1	1	1	3				3	
32	1	1	1	3				3	
33	1	1	1	3				3	
34	1	1	1	3				3	
35	1	1	1	3				3	
36	1	1	1	3				3	
37	1	1	1	3				3	
38	1	1	1	3				3	
39	1	1	1	3				3	
40	1	1	1	3				3	
41	1	1	1	3				3	
42	1	1	1	3				3	
43	1	1	1	3				3	
44	1	1	1	3				3	
45	1	1	1	3				3	
46	1	1	1	3				3	
47	1	1	0	2			2		FP & TV
48	1	1	1	3				3	
49	1	1	1	3				3	
50	1	1	1	3				3	
51	1	0	1	2			2		FP & TP
52	0	0	0	0	0				
53	0	0	0	0	0				
54	1	0	1	2				3	
55	1	1	1	3				3	
56	1	1	1	3				3	
57	1	1	1	3				3	
58	1	1	1	3				3	
59	1	1	1	3				3	
60	1	1	1	3				3	
61	1	1	1	3				3	
62	1	1	1	3				3	
63	1	1	1	3				3	
64	1	1	0	2			2		FP & TV
65	1	1	1	3				3	
66	1	0	0	1		1			FP
67	1	1	1	3				3	
68	1	1	0	2			2		FP & TV
69	1	1	0	2			2		FP & TV
70	1	1	1	3				3	
71	1	1	1	3				3	
72	1	1	0	2			2		FP & TV
73	1	1	1	3				3	

Table 5: Comparison of Faults Found by Three Testing Techniques (1 of 4).

F#	FP	TV	TP	Sum	ZER	ONE	TWO	THR	
	134	129	120	383	60	7	16	115	
74	1	1	0	2			2		FP & TV
75	1	1	1	3				3	
76	1	1	1	3				3	
77	1	1	1	3				3	
78	1	1	0	2			2		FP & TV
79	1	1	1	3				3	
80	0	1	0	1		1			TV
81	1	1	1	3				3	
82	1	1	1	3				3	
83	1	1	1	3				3	
84	1	1	0	2			2		FP & TV
85	1	1	1	3				3	
86	1	1	1	3				3	
87	1	1	1	3				3	
88	1	1	1	3				3	
89	0	0	0	0	0				
90	1	1	1	3				3	
91	1	1	1	3				3	
92	1	1	1	3				3	
93	1	1	1	3				3	
94	1	1	1	3				3	
95	1	1	1	3				3	
96	1	1	1	3				3	
97	1	1	1	3				3	
98	1	1	1	3				3	
99	1	1	1	3				3	
100	1	1	0	2			2		FP & TV
101	1	1	1	3				3	
102	1	1	1	3				3	
103	1	1	1	3				3	
104	1	1	1	3				3	
105	1	1	1	3				3	
106	1	1	1	3				3	
107	1	1	1	3				3	
108	1	1	1	3				3	
109	1	1	1	3				3	
110	1	1	1	3				3	
111	1	1	1	3				3	
112	1	1	1	3				3	
113	1	1	1	3				3	
114	1	1	1	3				3	
115	1	1	1	3				3	
116	1	1	1	3				3	

Table 5: Comparison of Faults Found by Three Testing Techniques (2 of 4).

F#	FP	TV	TP	Sum	ZER	ONE	TWO	THR	
	134	129	120	383	60	7	16	115	
117	1	1	1	3				3	
118	1	0	0	1		1			FP
119	1	1	1	3				3	
120	1	1	0	2			2		FP & TV
121	1	1	1	3				3	
122	1	1	1	3				3	
123	1	1	0	2			2		FP & TV
124	1	1	1	3				3	
125	1	1	1	3				3	
126	1	1	1	3				3	
127	1	1	0	2			2		FP & TV
128	1	0	1	2			2		FP & TP
129	1	1	1	3				3	
130	1	1	1	3				3	
131	0	1	1	2			2		TV & FP
132	1	1	1	3				3	
133	1	1	1	3				3	
134	1	1	1	3				3	
135	1	1	1	3				3	
136	0	0	0	0	0				
137	1	0	0	1		1			FP
138	1	1	1	3				3	
139	1	1	1	3				3	
140	0	0	0	0	0				
141	1	1	1	3				3	
142	1	0	0	1		1			FP
143	1	1	1	3				3	
144	1	1	1	3				3	
145	1	1	1	3				3	
146	1	1	1	3				3	
147	1	1	1	3				3	
148	1	1	1	3				3	
149	0	1	0	1		1			TV
150	1	1	1	3				3	
151	0	0	0	0	0				
152	1	1	1	3				3	
153	1	1	1	3				3	
154	1	1	1	3				3	
155	0	0	0	0	0				
156	0	0	0	0	0				
157	0	0	0	0	0				
158	0	0	0	0	0				
159	1	1	1	3				3	

Table 5: Comparison of Faults Found by Three Testing Techniques (3 of 4).

F#	FP	TV	TP	Sum	ZER	ONE	TWO	THR	
	134	129	120	383	60	7	16	115	
160	1	1	1	3				3	
161	1	1	1	3				3	
162	1	1	1	3				3	
163	1	1	1	3				3	
164	0	0	0	0	0				
165	1	1	1	3				3	
166	1	1	1	3				3	
167	0	0	0	0	0				
168	0	0	0	0	0				
169	1	1	1	3				3	
170	0	0	0	0	0				
171	0	0	0	0	0				
172	0	0	0	0	0				
173	1	1	1	3				3	
174	1	1	1	3				3	
175	1	1	1	3				3	
176	1	1	1	3				3	
177	1	1	1	3				3	
178	0	0	0	0	0				
179	1	1	1	3				3	
180	0	0	0	0	0				
194	1	1	1	3				3	
195	1	1	1	3				3	
196	0	0	1	1		1			TP
197	0	0	0	0	0				
198	1	1	1	3				3	
	134	129	120	383	60	7	16	115	

Table 5: Comparison of Faults Found by Three Testing Techniques (4 of 4).

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 tool and the experimentation that was carried out has well satisfied this goal. As a future goal, I'd like to recommend the following idea. Thus far, all testing has been based on functional transitions between states and modes in a state-based specification description of the software. However, states also have **data interactions**, and these interactions are unlikely to be adequately tested using these techniques. Thus, I suggest developing a **data-flow testing criterion** for state-based specifications, similar to traditional code-based data flow. Definitions and uses of variables would be identified in the states of the finite state machine, and a data flow graph could be developed from them.

References

[ABM98] Paul E. Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. In *Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 46–54, Brisbane, Australia, December 1998.

[ADO00] Aynur Abdurazik Paul Ammann, Wei Ding, and Jeff Offutt. Evaluation of three specification-based testing criteria. In *Proceedings of the Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '00)*, pages 179–187, Tokyo, Japan, September 2000.

[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.

[BB97] Mark Blackburn and Robert Busser. Automatic generation of test vectors for SCR-style specifications. In *Proceedings of the 1997 Annual Conference on Computer Assurance (COMPASS 97)*, pages 54–67, Gaithersburg MD, June 1997. IEEE Computer Society Press.

[Cor98] Rational Software Corporation. *Rational Rose 98: Using Rational Rose*. Rational Rose Corporation, Cupertino CA, 1998.

[GWZ94] A. Goldberg, T. C. Wang, and D. Zimmerman. Applications of feasible path analysis to program testing. In *Proceedings of the 1994 International Symposium on Software Testing, and Analysis*, pages 80–94, Seattle WA, August 1994.

[HKL97] C. Heitmeyer, J. Kirby, and B. Labaw. Tools for formal specification, verification, and validation of requirements. In *Proceedings of the 1997 Annual Conference on Computer Assurance (COMPASS 97)*, pages 35–47, Gaithersburg MD, June 1997. IEEE Computer Society Press.

[HL92] J. R. Horgan and S. London. ATAC: A data flow coverage testing tool for C. In *Proceedings of the Symposium of Quality Software Development Tools*, pages 2–10, New Orleans LA, May 1992.

[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.

[Mye79] G. Myers. *The Art of Software Testing*. John Wiley and Sons, New York NY, 1979.

[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.

[Off00] A. J. Offutt. Generating test data from requirements/specifications: Phase III final report. Technical report ISE-TR-00-02, Department of Information and Software Engineering, George Mason University, Fairfax VA, January 2000. http://www.ise.gmu.edu/techrep.

[OP97] A. J. Offutt and J. Pan. Detecting equivalent mutants and the feasible path problem. *The Journal of Software Testing, Verification, and Reliability*, 7(3):165–192, September 1997.

[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.

Contents

1 INTRODUCTION **2**

2 SUMMARY OF PHASES I, II AND III **3**
2.1 Summary of Phase IV Goals 4
2.2 Publications From This Project 4

3 TEST DATA GENERATION TOOL **6**
3.1 Assumptions 7
3.2 Architecture 8
3.3 Decisions in Algorithms 11
3.4 Results 13
3.4.1 Aircraft_Data_Sources - Table 7 13
3.4.2 Autopilot_Modes_Disengaged_Submode - Table 12 13
3.4.3 Autopilot_Modes_Engage 14
3.4.4 Autopilot_Modes_Engaged_Submode - Table 17 14
3.4.5 Flight_Modes_Flight_Director_Cues - Table 31 14
3.4.6 Flight_Modes_Flight_Director_Mode - Table 33 15
3.4.7 Flight_Modes_Lateral_Active - Table 34 16
3.4.8 Flight_Modes_Lateral_Approach - Table 40 17
3.4.9 Flight_Modes_Lateral_Navigation - Table 41 17
3.4.10 Flight_Modes_Lateral_Roll - Table 42 18
3.4.11 Flight_Modes_Vertical_Active - Table 44 19
3.4.12 Flight_Modes_Vertical_Altitude_Select - Table 45 19
3.4.13 Flight_Modes_Vertical_Approach - Table 47 20
3.4.14 Flight_Modes_Vertical_Flight_Level_Change - Table 50 21

4 EMPIRICAL EVALUATION **23**
4.1 Transition-pair Tests 23
4.2 Faults Inserted Into FGS 24
4.3 Results and Analysis 25

5 CONCLUSIONS **27**

6 FUTURE WORK **32**