# AUTOMATED MODEL ANALYSIS AND TEST GENERATION FOR FLIGHT GUIDANCE MODE LOGIC

*Robert D. Busser, Mark R. Blackburn, Ph.D., Aaron M. Nauman*
*Software Productivity Consortium/T-VEC Technologies, Herndon, VA*

## Introduction

Incomplete, ambiguous, or rapidly changing requirements can have a profound impact on the quality and cost of software development. In an effort to provide a more rigorous approach to flight-critical system development, Rockwell Collins used a formal specification modeling approach to develop the mode control logic of a Flight Guidance System (FGS) for a General Aviation class aircraft [7; 8]. Rockwell Collins later used an early version of Test Automation Framework (TAF) approach for model-based analysis and test automation to analyze the requirement model and generate tests for a new implementation of the FGS system [10].

The TAF approach integrates various government and commercially available model development and test generation tools to support defect prevention and automated testing of systems and software. Current implementations of the TAF have been demonstrated to reduce cycle time by 50 percent and increase quality by eliminating requirement defects and automating test [11]. The latest version of the TAF has been re-applied to the FGS system, and has uncovered numerous model contradiction (i.e., requirement defects) and implementation faults that went undiscovered in the analyses prior to 1998.

This paper describes the TAF model-based verification approach. It summarizes the new model and implementation errors that have been discovered. It briefly describes how the TAF approach can be used to locate requirement defects early in the development process, reduce manual test development effort, and reduce rework. It describes how the use of model-based development and test automation can be effectively used in the development and verification of systems that must meet the highest standards of safety, reliability, and quality.

## Background

Miller from Rockwell Collins used the CoRE [6] and SCR [5] methods to specify the requirements for the mode logic of an FGS. An FGS compares the measured state of an aircraft (position, speed, and attitude) to the desired state and generates pitch and roll guidance commands to minimize the difference between the measured and desired state. The mode logic accepts commands from the flight crew and a variety of systems such as the Flight Management System (FMS).

As reflected in Figure 1, the FGS was first specified by hand using the CoRE method, inspected, then entered into a tool supporting the SCR method provided by the Naval Research Laboratory (NRL). Despite careful review and correction of 33 errors in the CoRE model, the SCRtool's analysis capabilities revealed an additional 27 errors [8]. Statezni later used an early TAF translator [2] and the T-VEC [1] toolset to analyze the SCR model, generate test vectors and test drivers. The test drivers were executed against a java implementation of the FGS requirements [10] and revealed six errors. Offutt applied his tool to the FGS model and found two errors [9], and the latest TAF toolset, described in this paper, identified 25 errors.

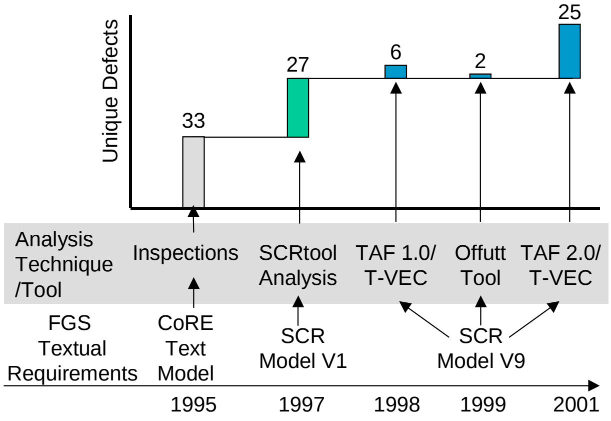


**Figure 1. Model Evolution and Analysis**

## Approach and Toolset

This section summarizes the TAF approach used to verify the FGS system in this latest effort.

### *Process Overview*

A conceptual process flow that relates the TAF artifacts and tools is shown in Figure 2.

- FGS specification was modeled using the SCRtool
- SCR-to-T-VEC translator (SCR2TVEC), developed by the Consortium and T-VEC, was used to translate the SCR model into a T-VEC test specification
- T-VEC tools were used on the T-VEC representation of the model requirements to automatically generate test vectors (i.e., test cases with test input values, expected output values and traceability information) and requirement-to-test coverage metrics
- T-VEC automatically generated test drivers to execute tests against the FGS Java code
- Test results were compared with the test vector expected outputs, and a results report was produced

### *SCR Concepts and Tool*

SCR is a table-based modeling approach, as shown in Figure 3, for modeling system and software requirements. SCR represents system inputs as **monitored variables**, system outputs as **controlled variables** and intermediate values as **term variables**. Variables are defined as primitive types (e.g., Integers, Float, Boolean) or as user-defined types including enumerations. Behavior is defined using a tabular approach relating four model elements: modes, conditions, events, and terms. The functionality or behavior of the system is defined using tables to relate monitored variables to controlled variables. There are three basic types of tables (with two variants):

- Condition table (with mode or modeless)
- Event table (with mode or modeless)
- Mode transition table for a mode class

A **mode class** is a state machine, where system states are called system modes and the transitions of the state machine are characterized by guarded events. A **condition** characterizes system state with an expression that evaluates to true or false. An **event** occurs when any system entity changes value.

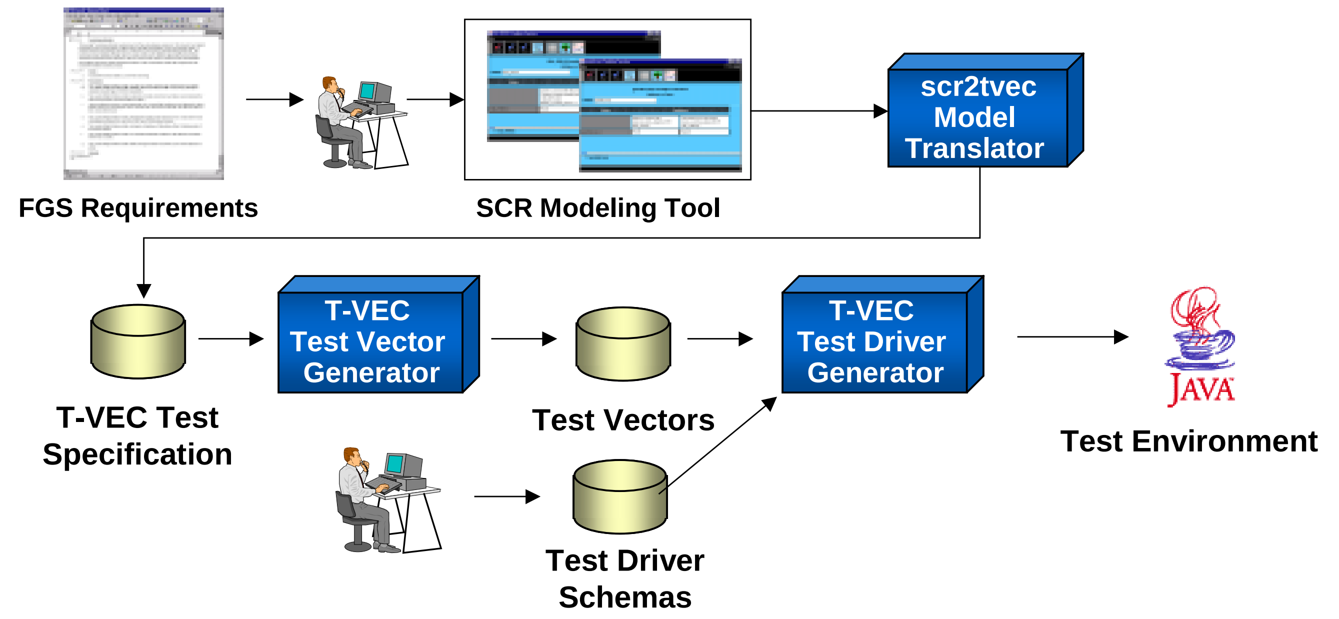


Figure 2. Process Flow and Artifacts

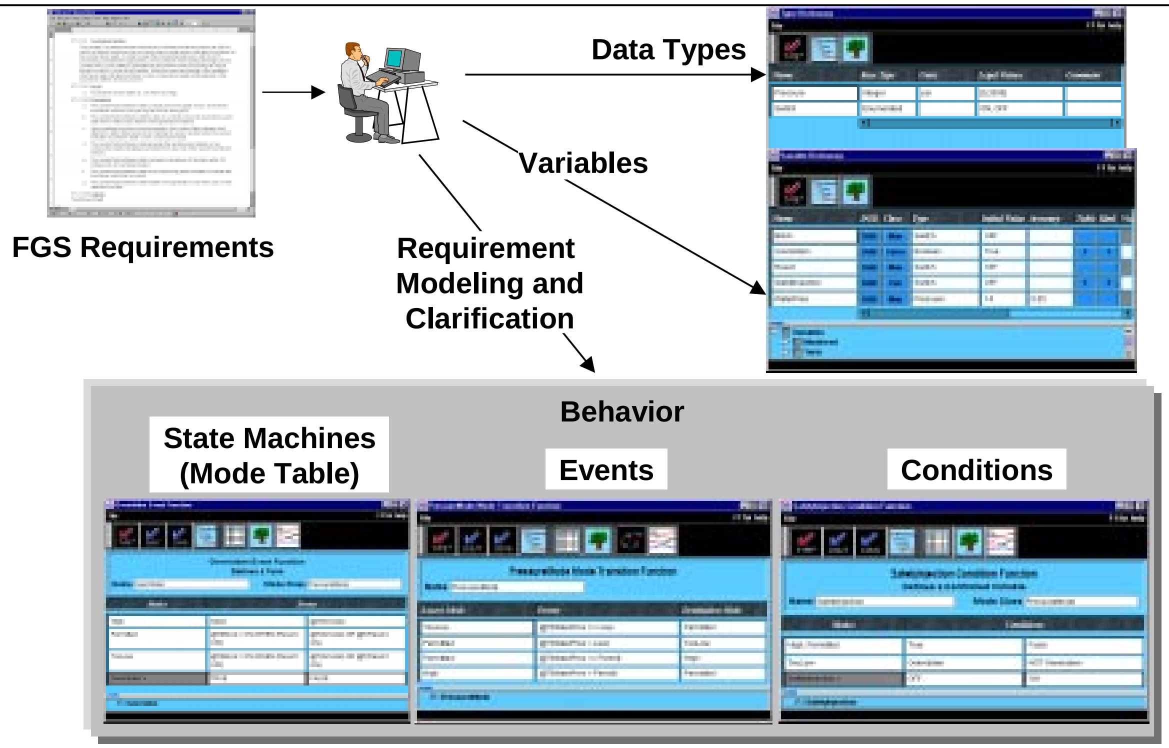

FGS Requirements
Requirement Modeling and Clarification
Data Types
Variables
Behavior
State Machines (Mode Table)
Events
Conditions

**Figure 3. SCR Modeling Constructs**

SCR modeling permits condition, event and mode tables to be combined. Terms and controlled variables are functions of input variables, modes, or other terms. Their values are defined in the model through event or condition tables. This allows complex relationships between monitored and controlled variables to be described using term variables describing simpler and potentially reusable relationships, as illustrated in Figure 4.

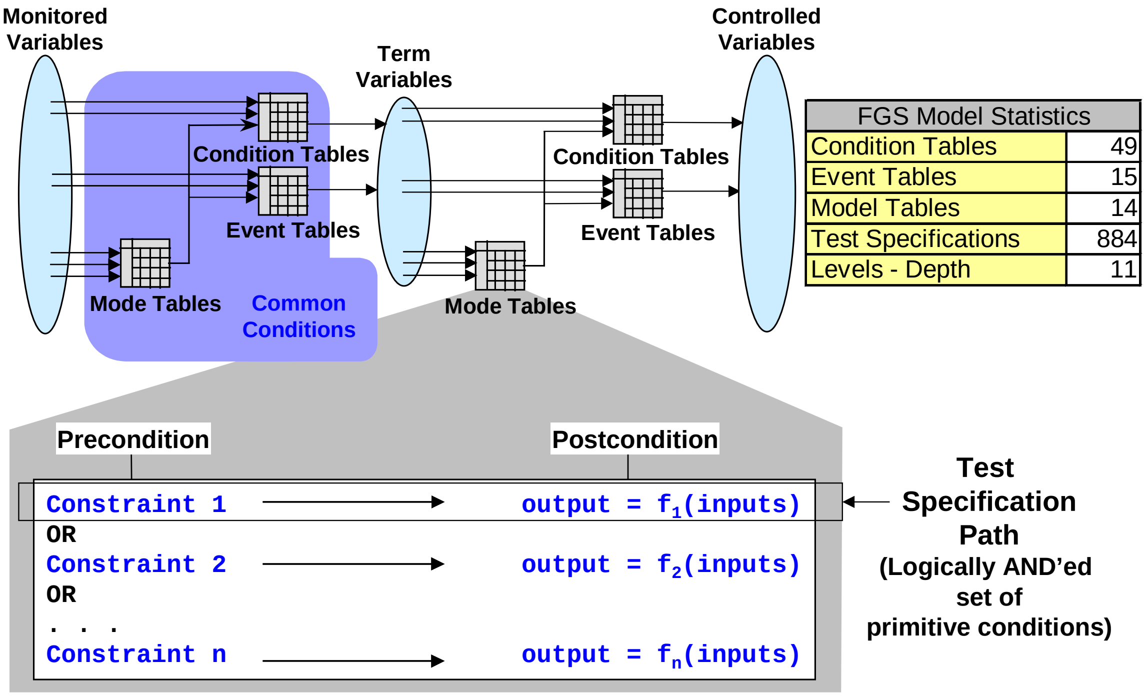
### *Model Translations*

The TAF translator converts SCR models, which are composed of condition, event, and mode tables into T-VEC test specification models. Each SCR table is represented by a T-VEC subsystem. From each subsystem the T-VEC toolset produces a set of precondition/postcondition pairs, referred to as ***test specification paths (TSP)***, to support model analysis and test generation. The TSPs are hierarchical and fully represent cross-table dependencies in the SCR model that enables locating defects related to more than one SCR table.

### *Test Generation and Defect Identification*

Test vector generation attempts to produce a test vector for every TSP. A ***test vector*** is a set of test input values that satisfy the input constraints, and an expected output value that is derived by evaluating the postcondition with the input values [1]. Informally, from a test generation perspective, a specification is ***satisfiable*** if at least one test vector exists for every TSP [2]. If a test vector is not produced for a TSP, it probably contains a contradiction (a requirement defect).

The SCRtool can check consistency of individual tables, but many inconsistencies result from cross-table dependency relationships that are analogous to feature interaction problems. Using TAF to identify these cross-table defects is a two-step process: 1) the test vector generator attempts to find a test for every TSP, 2) a post-processing activity identifies TSPs that have no associated test vector. The TSPs with no associated test vectors are traced back to the requirements model to identify requirement defects.



| FGS Model Statistics | |
|---|---|
| Condition Tables | 49 |
| Event Tables | 15 |
| Model Tables | 14 |
| Test Specifications | 884 |
| Levels - Depth | 11 |

**Figure 4. Representation of Test Specification Model**

### *Test Drivers, Execution and Results Analysis*

Test driver generation automates the time consuming and error prone activity sometimes referred to as test script development. The T-VEC test driver generator combines test vectors and a test driver schema to produce a test driver (script) and a file of expected test outputs. The test vectors describe the test data, while the test driver schema describes a generic algorithm for executing tests in a specific environment. Schemas are typically defined once per test environment and include the following high level operations: initialize the system, set system outputs to value other than the expected result, set system input values, execute the system under test, retrieve and store the actual test outputs.

Results analysis compares the actual results of test execution to expected test results as defined by the test vector expected outputs. A comparator utility supplied with the T-VEC tools supports automating the results comparisons while accounting for any numeric tolerances.

## Model and Implementation Defects

As a result of the prior analysis using different verification tools and techniques [7; 8; 10] the FGS specification was believed to have no remaining defects. The implementation of the specification written in Java was known to have six documented coding errors. This section presents the results of applying of the second generation of the TAF translator and T-VEC toolset on the FGS specification. It includes a description of the new model and implementation defects identified with the new tools.

### *Model Analysis*

The FGS model contains 78 SCR tables including 47 condition tables, 14 mode transition tables, and 15 event tables. Using the default scr2tvec translation options, the FGS model was translated into 78 primary T-VEC subsystems each corresponding to an SCR table. The translator also produced two additional T-VEC subsystems that package data types and constants used throughout the model.

**Table 1. Analysis Details**

| Defect Description | Defect Type | Unique Defects | Total Defects |
|---|---|---|---|
| Invalid event expression for related mode table | Model | 5 | 7 |
| Invalid constraint for dependency | Model | 5 | 18 |
| Mode transition implemented incorrectly | Code | 2 | 3 |
| Specification not implemented | Code | 1 | 1 |
| Variable referenced before set | Code | 3 | 26 |
| Incorrect implementation of mode logic | Code | 1 | 4 |
| Incorrect code (likely cut/paste error) | Code | 1 | 1 |
| Incorrect event implementation | Code | 3 | 44 |
| Hidden bug - coincidental correctness | Code | 1 | 2 |
| Unmodeled domain knowledge | Code | 3 | 8 |
| New Errors |  | 25 | 114 |
| Known bugs | Code | 6 | 6 |
| Total Errors Detected |  | 31 | 120 |

### *Analysis Summary*

Translation and processing by the T-VEC tools produced 884 unique TSPs. The T-VEC test generation system uses a test selection heuristic based on domain testing theory where low-bound and high-bound values are selected for each constraint.[1] Domain testing theory is based on the intuitive idea that faults in implementation are more likely to be found by test points chosen near appropriately defined program input and output domain boundaries [12]. By default, T-VEC attempts to determine two test vectors for each TSP, one with low-bound values and another with high-bound values. Therefore, test generation should have produced 1778 test vectors. However, due to latent errors remaining in the FGS SCR specification, only 1700 test vectors were produced.

Table 1 summarizes the classes of defects identified. The first two rows indicate 10 newly discovered model contradictions resulted in 25 model defects. The missing test vectors discussed in the previous section occurred because these contradictions produce unsatisfiable TSPs. Each contradiction involved at least one event or condition table and at least one mode table. An additional 21 faults of various types in the implementation resulted in 95 test failures. The test cases also revealed six known bugs. These defects are described in the following subsections.

### *Defect Types*

As defined by Howden and later refined by Zeil, there are two types of implementation errors: computation errors and domain errors. A **computation error** occurs when the correct path through the program is taken, but the output is incorrect due to faults in the computation along the path. A **domain error** occurs when an incorrect output is generated due to executing the wrong path through a program [How76; Zei89]. Both domain and computation errors can also occur in models.

### *Model Defects*

The concept of a program path and its related output computation is analogous to a TSP from a translated model. A domain error for a TSP means that there is no input set that satisfies the test specification's constraints. Consider the trivial example (and graphic insert):

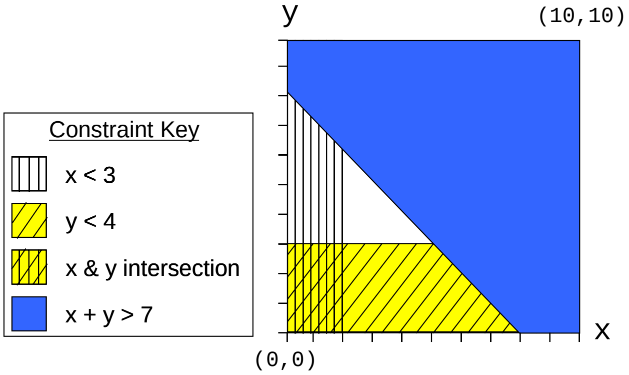
```
x: Integer with domain from 0 to 10
y: Integer with domain from 0 to 10
z: Integer with domain from 0 to 10
```

If there is a requirement that specifies

```
z = 0 when
      x < 3 AND
      y < 4 AND
      x + y > 7
```

[1] White and Cohen proposed domain testing theory as a strategy to select test points to reveal domain errors [13]. Their theory is based on the premise that if there is no coincidental correctness, then test cases that localize the boundaries of domains with arbitrarily high precision are sufficient to test all the points in the domain.

```
then
   maximum value for x is 2
   maximum value for y is 3
   minimum value for x + y is 8
```



There are no values of x and y that satisfy this requirement. This is illustrated in the figure as there is no intersection of the three regions defined by x < 3, y < 7 and x + y > 7. The constraint expression cannot be satisfied and is therefore contradictory. The contradiction is a domain error, because the variable z can never be assigned a value of 0 through this requirement. Thus, the requirement is untestable. Real-world problems typically include complex constraints that span many modules or components of an application. In these situations it can be difficult to isolate these errors through manual processes. Automated model analysis provides a tool for locating these errors. T-VEC identifies arithmetic contradictions for expressions including all primary scalar data types and common mathematic operators.

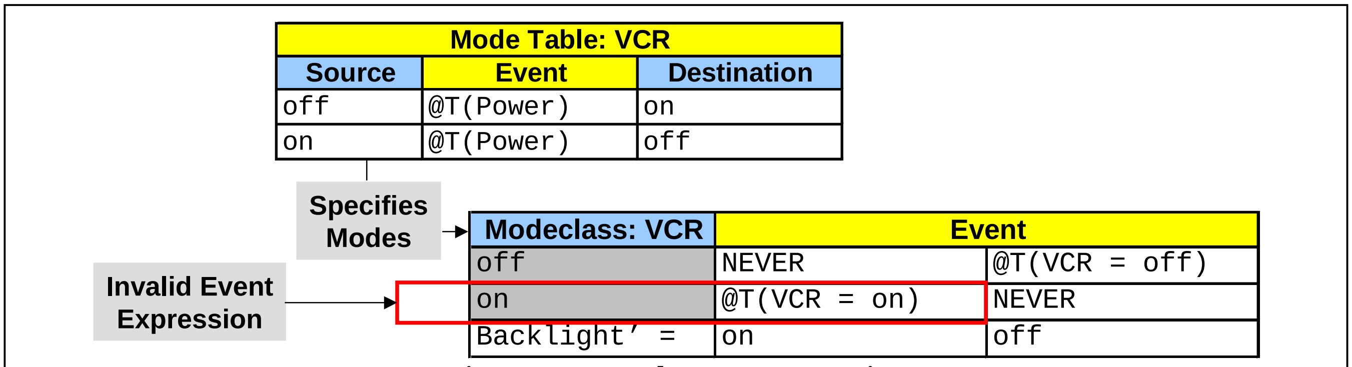
### *Specification Contradictions*

Analysis of the T-VEC vector generation results revealed 15 subsystems that included specification contradictions preventing production of a test vector for one or more of the subsystem’s TSPs. Two types of contradictions were identified. One type included invalid event specifications in an event table that was dependent on a mode class. The second type of defect involved conditions within a condition table that were not satisfiable with respect to dependencies on a model class. Domain knowledge is required to understand the contradictions within the FGS model, therefore the following two examples illustrate the essence of these contradictions.

### *Invalid Event Expression*

Five event tables contain contradictions. Each of these tables is mode-dependent meaning each event is dependent on the system being in a specific mode. The error in each table is similar to the one illustrated in the VCR example shown in Figure 5. The VCR mode table indicates that if the VCR is in the *off* mode and the Power button is pressed, that the VCR will go to the *on* mode, and in the *on* mode if the Power button is pressed, it will go to the *off* mode. The VCR’s LCD display (Backlight) should be turned on and off when the VCR is turned on and off. The Backlight event table attempts to specify this behavior, but it includes a subtle inconsistency between the mode dependency and event expression.

In a mode dependent SCR event table, events are triggered only when the system is in the mode associated with the event expression. As highlighted in Figure 5, the event associated with turning the VCR on, @T(VCR = on), is triggered only when the VCR is on.. The event expression @T(VCR = on) means that previously the expression VCR=on was FALSE and now it is TRUE. The transition of the event expression from



| Mode Table: VCR | | |
|---|---|---|
| Source | Event | Destination |
| off | @T(Power) | on |
| on | @T(Power) | off |

| Modeclass: VCR | Event | |
|---|---|---|
| off | NEVER | @T(VCR = off) |
| on | @T(VCR = on) | NEVER |
| Backlight’ = | on | off |

**Figure 5. Example Event Expression**

FALSE to TRUE defines the point in time when the event is “triggered.”

In this case, this transition can never occur as specified, because the mode dependency requires that the VCR remain on (i.e., in the *on* mode both previously and presently) while the event specifies that the VCR was previously not *on*. Thus, the table includes a contradiction.

### *Invalid Constraint for Dependency*

The second type of model defect involves conditions that cannot be satisfied in the context of a mode dependency.

This example shown in Figure 6 contains two tables that are related to a mode table (not shown) for the VCR. The valid VCR modes include: play, ff (fast forward), rew (rewind), rec (record), play and prog (program). The first table defines the display color for the VCR LCD text output, and the second table defines the values of the Text. The Display_Color table indicates the color is set to *none* when the Text output is *Blank* in any Operation mode. The contradiction occurs because the Text term table indicates that Text is only *Blank* in the *prog* Operation mode. Therefore, the highlighted conditions in the Display_Color table can never be satisfied, because Text is never *Blank* when the operation mode is *play*, *ff*, *rew*, or *rec*. Five new instances of this type of defect were identified in the FGS model that resulted in 18 unsatisfiable TSPs.

### *Overlapping Conditions / Non-Determinism*

A set of model tables included constraints with overlapping conditions. Such overlaps make the model non-deterministic. These problems were identified when the generated tests did not pass (i.e., the expected output and the actual output did not match). These situations are discussed in the next section.

## Verification of FGS Implementation

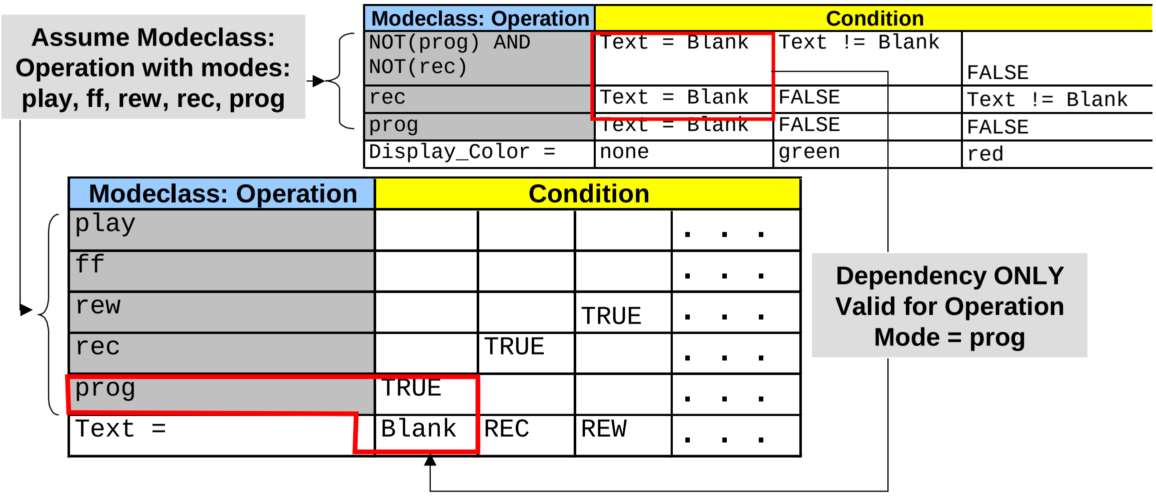
The T-VEC test driver generator produces test drivers for each of the test vectors derived from the model. The test drivers were executed against a Java implementation of the FGS model. The first generation tools helped identify six implementation errors. The updated version of the tools uncovered an additional 21 new faults manifesting in 95 test failures. These implementation faults have also been classified.

### *Mode Transition Implemented Incorrectly*

Two faults, resulting in three test failures, were related to if-then-else conditions that did not correctly implement the associated mode transition constraints.

### *Specification Not Implemented*

The update method for a term that was dependent on four other table specifications was not fully implemented. This was likely caused by the complexity associated with multiple levels of table



Assume Modeclass: Operation with modes: play, ff, rew, rec, prog

| Modeclass: Operation | Condition | | |
|---|---|---|---|
| NOT(prog) AND NOT(rec) | Text = Blank | Text != Blank | FALSE |
| rec | Text = Blank | FALSE | Text != Blank |
| prog | Text = Blank | FALSE | FALSE |
| Display_Color = | none | green | red |

| Modeclass: Operation | Condition | | | |
|---|---|---|---|---|
| play | | | | . . . |
| ff | | | | . . . |
| rew | | | TRUE | . . . |
| rec | | TRUE | | . . . |
| prog | TRUE | | | . . . |
| Text = | Blank | REC | REW | . . . |

Dependency ONLY Valid for Operation Mode = prog

**Figure 6. Example of Invalid Condition for Table Reference**

dependencies typical of complex systems. One fault of this type manifested in one failure.

### *Variable Referenced Before Set*

Three faults, resulting in 26 failures, were related to incorrect logic that allowed variables to be referenced before set.

### *Incorrect Implementation Of Mode Logic*

The logic implementing event transitions of a mode table assumed a dependency ordering, however, the SCR model is a declarative model and the order of the modes and event transitions within the model has no relevance. One fault of this type manifested in one failure.

### *Incorrect Code (likely cut/paste error)*

After analyzing the code, the most likely cause of the one failure appears to be a cut/paste error.

### *Incorrect Event Implementation*

Although the order of the events in an event table has no explicit meaning, event tables are often implemented using sequences of if-then-elseif-else statements that have order. Three faults related to mapping the SCR declarative (i.e., unordered) model into this ordered logic, resulted in 44 test failures.

### *Hidden Bug (coincidental correctness).*

A general test driver pattern is typically employed to ensure that output objects are set to a value other than the expected output to avoid coincidental correctness, which can occur by the output retaining the expected output from system initialization or prior execution of the code. In this case, the output was not set to a value other than expected and the fault was therefore not detected. Fixing this initialization revealed an error not discovered previously.

### *Unmodeled Domain Knowledge*

The FGS model includes a type of “momentary contact” switch that automatically returns to the OFF position whenever activated. The model assumed this behavior and did not model it explicitly. This resulted in an inconsistency between the implementation and model, which resulted in a eight test failure associated with three faults.

## Summary

This paper describes the ongoing effort to evolve model analysis for complex systems to support development of high reliability systems, especially safety critical ones. The results illustrate the importance of using tools to support the analysis of complex systems. This historical evolution of the original FGS model used inspections to remove model defects and the SCRtool model analysis capabilities to identify problems in individual tables. The first generation TAF/T-VEC tools and Offutt tool were able to detect additional faults including those related to multiple tables. The latest version of the TAF/T-VEC tools identified many model defects that primarily involved multiple tables. In addition, these tools helped uncover many additional faults in the implementation. Model defects and implementation faults similar to those discovered occur commonly in complex systems. As software-based systems continue to evolve, the capabilities demonstrated can provide greater assurance that these systems operate dependably.

### *Other Applications and Results*

The core capabilities underlying this approach were developed in the late 1980s and proven through use in support of FAA certifications for flight critical avionics systems [1]. Statezni described how the approach supports requirement-based test coverage mandated by the FAA with significant life cycle cost savings [10; 11]. Safford presented results stating the approach reduced cost, effort, and cycle-time by eliminating requirement defects and automating testing [11]. Safford’s presentation summarized the benefits:

- Better quality requirements for design and implementation help eliminate rework in those phases as well as during test
- Verification modeling can reduce the time normally spent in verification test planning by up to 50 percent

- Test generation from a verification model can eliminate up to 90 percent of the manual test creation and debugging effort
- Both the number of test cases and the phasing of their execution can be optimized, eliminating test redundancy
- A known level of requirements coverage can be planned, and measured during test execution

The approach and tools described in this paper have been used for modeling and testing system, software integration, software unit, and some hardware/software integration functionality. It was used to identify the most likely cause of the Mars Polar Lander crash [4]. It has been used for functional security testing [3], as well as, critical applications like telemetry communication for heart monitors, flight navigation, guidance, autopilot logic, display systems, flight management and control laws, airborne traffic and collision avoidance. In addition, it has been applied to non-critical applications such as workstation-based Java applications with GUI user interfaces and database applications. The approach supports automated test driver generation in a variety of open languages (e.g., C, C++, Java, Ada, Perl, PL/I, SQL), as well as, proprietary languages, COTS test injection products, and test environments.

## References

[1] Blackburn, M.R., R.D. Busser, T-VEC: A Tool for Developing Critical System. In Proceeding of the Eleventh International Conference on Computer Assurance, Gaithersburg, Maryland, pages 237-249, June, 1996.

[2] Blackburn, M.R., R.D. Busser, J.S. Fontaine, Automatic Generation of Test Vectors for SCR-Style Specifications, In Proceeding of the 12th Annual Conference on Computer Assurance, Gaithersburg, Maryland, pages 54-67, June, 1997.

[3] Blackburn, M.R., R.D. Busser, A.M. Nauman, R. Chandramouli, Model-based Approach to Security Test Automation, In Proceeding of Quality Week 2001, June 2001.

[4] Blackburn, M.R., R. Knickerbocker, R. Kasuda, Applying the Test Automation Framework to the Mars Lander Touchdown Monitor, Lockheed Martin Joint Symposium 2001 (JS01), June 4-6, 2001.

[5] Heitmeyer, C., R. Jeffords, B. Labaw, Automated Consistency Checking of Requirements Specifications. ACM TOSEM, 5(3):231-261, 1996.

[6] Faulk, S.R., L. Finneran, J. Kirby, and A. Moini. Consortium requirements engineering guide-book. Technical Report SPC-92060-CMC, Software Productivity Consortium, 2214 Rock Hill Road, Herndon, VA 22070, December 1993.

[7] Steven P. Miller and Karl F. Hoech Specifying the Mode Logic of a Flight Guidance System in CoRE

[8] Steve Miller, Specifying the Mode Logic of a Flight Guidance System in CoRE and SCR. Second Workshop on Formal Methods in Software Practice (FMSP'98), Clearwater Beach, Florida, March, 1998.

[9] Offutt, A.J., Generating Test Data From Requirements/Specifications: Phase III Final Report, George Mason University, November 24, 1999.

[10] Statezni, David, Industrial Application of Model-Based Testing, 16th International Conference and Exposition on Testing Computer Software, June 14-18, 1999.

[11] Safford, Ed, L. Test Automation Framework, State-based and Signal Flow Examples, Twelfth Annual Software Technology Conference, 30 April - 5 May 2000.

[12] Tsai, W. T., D. Volovik, T. F. Keefe, Automated test case generation for programs specified by relational algebra queries, IEEE Transactions on Software Engineering, 16(3):316-324, March 1990.

[13] White, L.J., E.I. Cohen, A Domain Strategy for Computer Program Testing. IEEE Transactions on Software Engineering, 6(3):247-257,May, 1980.