# Using Models for Development and Verification of High Integrity Systems

Mark Blackburn, Robert Busser, Aaron Nauman, Travis Morgan
Software Productivity Consortium/T-VEC
2214 Rock Hill Road, Herndon, VA 20170

**Abstract**

*The Test Automation Framework (TAF) approach for model-based development and automatic test generation has been demonstrated to reduce cycle time by 50% and increase quality by eliminating requirement defects to reduce rework, and automating test. The paper describes organizational best practices for applying model-based testing of requirement and design-based models, and describes concepts of model defect analysis, and test sequences used to verify dynamic systems. Lastly, it describes approaches for applying model-based test generation tools with test driver generation and code coverage tools to support verification of high integrity software-intensive systems.*

## 1. Introduction

High integrity systems [WIK92] demand the highest degrees of dependability to support properties such as safety, security, reliability or availability. The increased complexity has led to increased cost of the verification process. While properties such as safety and security cannot be "tested" into a system, verification is still a critical activity in the process to provide assurance arguments for the dependability of the system.

Improvements in software-system testing are needed throughout industry. The National Institute of Science and Technology (NIST) estimated the cost of insufficient software testing processes in the United States during 2000 at $59 billion. It is common for software tests to be executed automatically, but the tests themselves are often created manually. This process relies on the judgment and experience of testing professionals. It is impossible in practice for such teams to anticipate and develop a test for every way that a program might fail. When done manually, the tasks related to test design are typically slow, error prone, and highly variable. They can account for 60 percent of testing effort. Organizations have reported spending nearly 50 percent of their test effort just developing and debugging test scripts. The difficulties of such test development are compounded because the testing process is often done under extreme time pressures. Significantly, testers find that about half of the defects found in the final code are due to defects in the underlying requirements [NCS99].

Requirement and design-based models are increasingly used in aircraft, medical and automotive software-system development where high integrity is demanded. The more rigorous modeling approaches support simulation and code generation, but have limited support for automated test generation. To address this need, the Test Automation Framework (TAF) approach for model-based analysis and test automation was developed. TAF integrates various government and commercially available model development and test generation tools to support defect prevention and automated testing of systems and software as shown in Figure 1. TAF supports modeling methods that focus on representing requirements, like the Software Cost Reduction (SCR) method, as well as methods that focus on representing design information, like Simulink® or MATRIXx®, which supports control system modeling for aircraft and automotive systems.

Through the use of model translation, requirement-based or design-based models are converted into a form where T-VEC, the test generation component of TAF, produces tests vectors. Test vectors include inputs as well as the expected outputs with requirement-to-test traceability information. T-VEC also supports test

driver generation, requirement test coverage analysis, and test results checking and reporting. The test driver mappings and test vectors are inputs to the test driver generator that produces test drivers. The test drivers are then executed against the implemented system during test execution. TAF is also integrated with requirement management tools, such as Telelogic's DOORS® to provide full traceability from a DOORS requirement to a generated test case. Lastly, TAF is also integrated with code coverage-based tools such as LDRA Testbed® that allows the generated tests to be measured for code-based tests coverage.

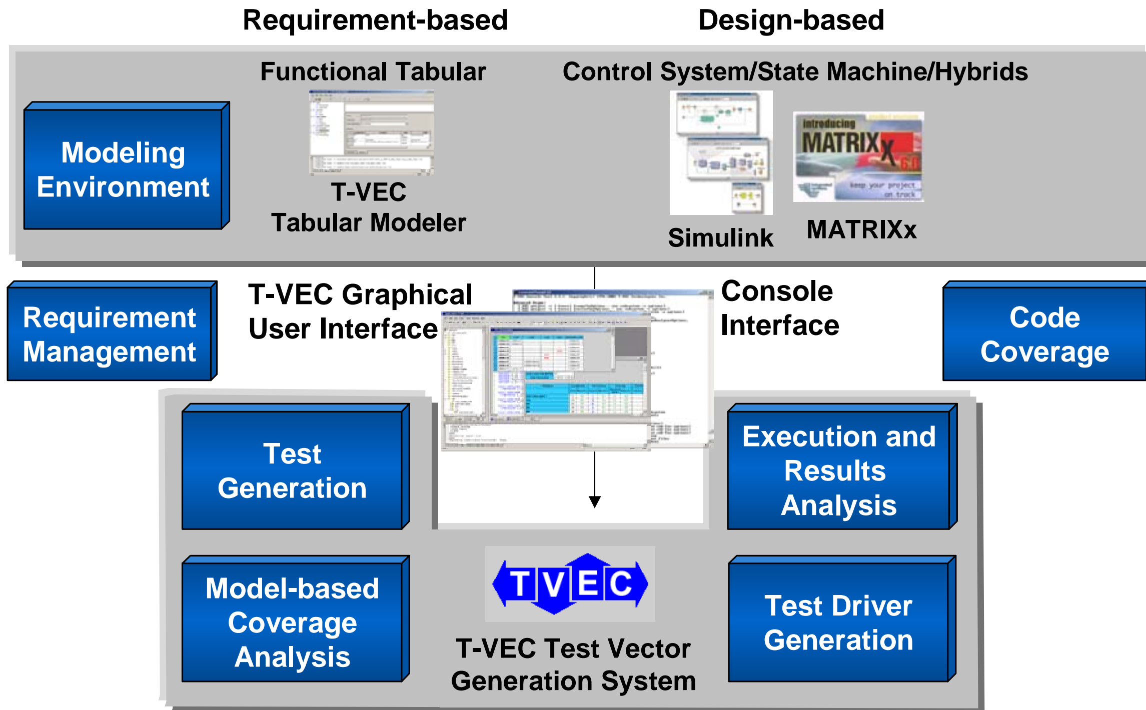



**Figure 1. TAF Integrated Components Background**

The core capabilities of TAF were developed in the late 1980s and proven through use in support of FAA certifications for flight critical avionics systems [BB96]. The approach supports requirement-based test coverage mandated by the FAA with significant life cycle cost savings [Sta99; Sta00; Saf00].

The process and tools described in this paper have been used for modeling and testing system, software integration, software unit, and hardware/software integration functionality. It has been applied to critical applications in medical and aerospace, supporting automated test driver generation in most languages (e.g., C, C++, Java, Ada, Perl, PL/I, SQL), as well as, proprietary languages, and test environments. The T-VEC tools have tool qualification packages that can be used to support FAA and FDA certifications. The qualification packages are compliant with FAA Software Approval Guidelines, Qualification Of Software Tools Using RTCA/DO-178B [DOT99].

## 2. Requirement-based Models and Automatic Test Generation

This section briefly describes the typical scenario for using the TAF to support requirement-based modeling and automatic test generation. A model is developed for a component's requirements and interfaces, and tests are generated from it. The test cases are then automatically transformed into test script (aka test drivers) for automated test execution. Test engineers work in parallel with requirement and design engineers to refine the requirements and model them to support automated test design and test execution. The following outlines the process, as depicted in Figure 2:

1. Working from whatever requirements artifacts are available, testers or modelers create models using a tool based on the SCR method [AFBPP92], such as the SCRtool [HJL96] or T-VEC Tabular Modeler (TTM). Tables in the model represent each output, specifying the relationship between input values and resulting output values. Models are automatically checked for inconsistencies. The tester interacts with the requirements engineers or analysts to validate the model as a complete and correct interpretation of the requirements.
2. The tester maps the variables (inputs and outputs) of the model to the interfaces of the system in object mappings. The nature of these interfaces depends on the level of testing performed. At the system level, the interfaces may include graphical user interface widgets, database APIs, or hardware interfaces. At the lowest level, they can include class interfaces or library APIs. The tester uses these object mappings with a test driver pattern to support automated test script generation. The tester works with the designers to ensure the validity of the interface mappings from model to implementation.
3. The T-VEC tool generates a set of test vectors for testing each (alternative) path in the model. These test vectors include test inputs and expected test outputs, as well as model-to-test traceability.
4. T-VEC generates the test drivers using the object mappings and schema. A schema is created once for each test environment. The schema defines the algorithmic pattern to carry out the execution of the test cases. The test driver executes in the target or host environment. The test drivers typically are designed as an automated test script that sets up the test inputs enumerated in each test vector, invokes the element under test, and captures the results.
5. Finally, T-VEC analyzes the test results. It compares the actual test results to the expected results and highlights any discrepancies in a summary report.

This conceptual process has been applied to modeling many different types of application in various application domains. More details on the process can be found in "Interface-Driven, Model-Based Test Automation," [BBN03].

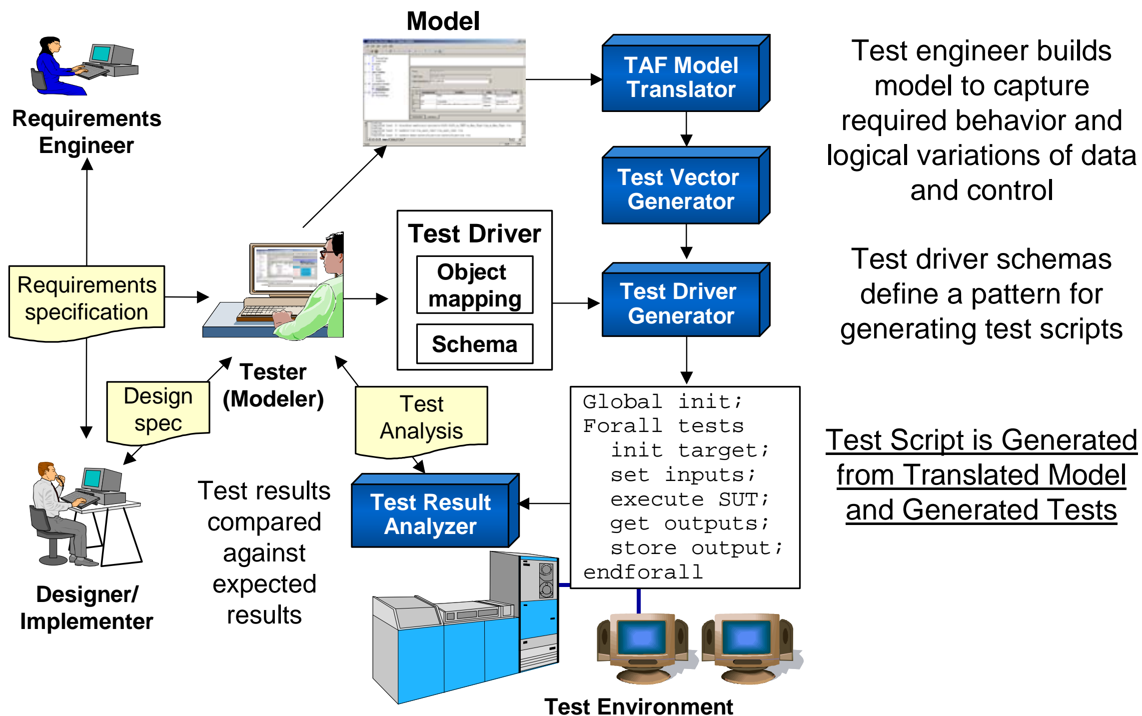



**Figure 2. Model-Based Test Automation**

## 2.1 Automated Test Generation, Execution, and Coverage

TAF uses the model to traverse the logical paths through the program, determining the locations of boundaries defined by constraints in the model and identifying reachability problems, where a particular thread through a model may not be achievable in the program itself. TAF uses test selection criteria based on domain testing and equivalence classes represented by the model constraints. These tests have been shown to uncover errors not identified by manually developed tests cases. The test vectors are used to verify the code that implements the model and to identify three main categories of model error:

- Mathematical error, e.g. division by a domain that spans zero, such as +/-1.0e+03; the division operation will be flagged as being a potential divide-by-zero hazard;
- Range overflow/underflow, i.e. signals which at some point in the model have values outside the specified bounds of the type of that signal;
- Logical contradiction, e.g. $(x > 0)$ & $(x < 0)$. Program errors often occur at boundaries or equivalence classes, logical points in software at which decisions are made [WJ91; WC80; How76; Zei89].

T-VEC generates input test vectors and predicts output results from the model. The tool subsequently generates a test harness to wrap the implementation code or interface to a system under test. Executing the test harness verifies that the input test vectors, when applied to the implementation, give the predicted output. Either an autocode generator or a programmer can provide the tested implementation code. T-VEC can provide test coverage measurements of the implementation in terms meaningful to the model.

Another benefit is that the test vectors can be exported to a dynamic test code tool, such as LDRA Testbed, to obtain coverage statistics that are meaningful measurements in implementation terms (e.g., 100% modified condition decision coverage (MC/DC) code coverage). Thus, the test vectors can be used for unit and integration test of the implementation and this can also provide further evidence to support the implementation verification.

## 2.2 Improved Requirements

Another unexpected benefit achieved is better understanding of the requirements, improved consistency, completeness, and most importantly, early requirement defect identification and removal. Models provide a means for stakeholders to better understand the requirements and assist in recognizing omissions. Tests automatically derived from the model support requirement validation through manual inspection or execution within simulation or host environments.

In order to be testable, a requirement must be complete, consistent and unambiguous. While any potential misinterpretation of the requirement due to incompleteness is a defect, TAF focuses on another form of requirement defect, referred to as a contradiction or feature interaction defect. These types of defects arise from inconsistencies or contradictions within requirements or between them. Such defects can be introduced when more than one individual develops or maintains the requirements. Often the information necessary to diagnose requirement contradictions spans many pages of one or more documents. Such defects are difficult to identify manually when requirements are documented in informal or semi-formal manners, such as textual documents. Although rigorous manual inspection techniques have been developed to minimize incompleteness and contradictions, there are practical limits to their effectiveness. These limits relate to human cognition and depend on the number and experience of people involved. TAF supports more thorough requirement testability analysis, allowing developers to iteratively refine and clarify models until they are free of defects.

Several companies, as described below have recognized how defect discovery using model-based test automation is both more effective and more efficient than using only manual inspection methods. One pilot study, conducted by a company, comparing formal Fagan inspections with TAF requirement verification, revealed that Fagan inspections uncovered 33 defects. In comparison, TAF uncovered all 33 of the Fagan inspection defects plus 56 more. Attempting to repeat the Fagan inspection did not improve its results. The improved defect detection of TAF prevented nearly two-thirds more defects from entering the rest of the development lifecycle.

Rockwell Collins had similar results when they applied TAF to a Flight Guidance System (FGS) for a General Aviation class aircraft [Mil98]. As reflected in Figure 3, the FGS was first specified by hand using the Consortium Requirement Engineering Method (CoRE). It was then inspected, and about a year later it 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. Statezni later used an early TAF translator and the T-VEC toolset to analyze the SCR model, generate test vectors and test drivers [Sta00]. The test drivers were executed against a java implementation of the FGS requirements and revealed six errors. Offutt applied his tool to the FGS model and found two errors [Off99]. The latest TAF toolset, described in this paper, identified 25 errors more than the original 27 errors.

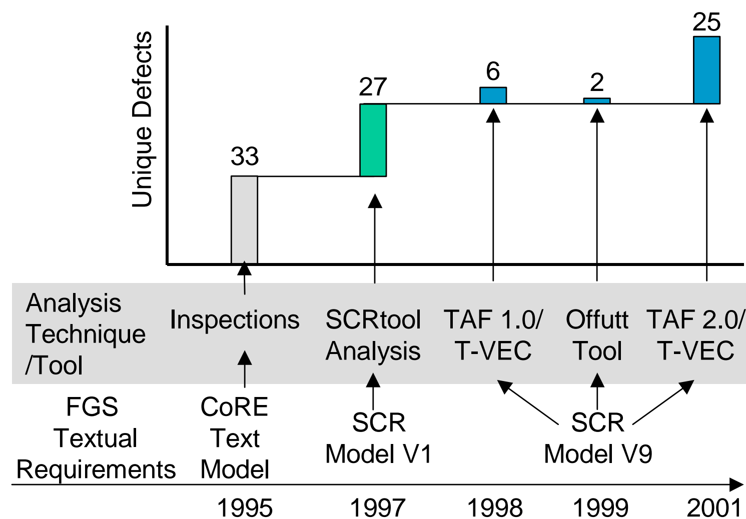



**Figure 3. Model Evolution and Analysis**

Following manual test generation practices, defects are not identified until late in the process, sometimes after release, when they are most expensive to fix. By automating test generation based on models, defects are found earlier in the process and faster. The rate of defect discovery increases early in the process, but quickly curtails. Many defects are found in the requirements phase, before they propagate to later development phases. Defect prevention is most effective during the requirements phase when it costs two orders of magnitude less than after the coding process.

Figure 4 represents the conceptual differences between manual and automatic test generation. The existing process of discovering and eliminating software defects is represented by the curve labeled "Old" while the effects of early defect discovery aided by automation is illustrated by the trend curve labeled "New." Industrial applications have demonstrated that TAF directly supports early defect identification and defect prevention through the use of requirement testability analysis [Saf00].

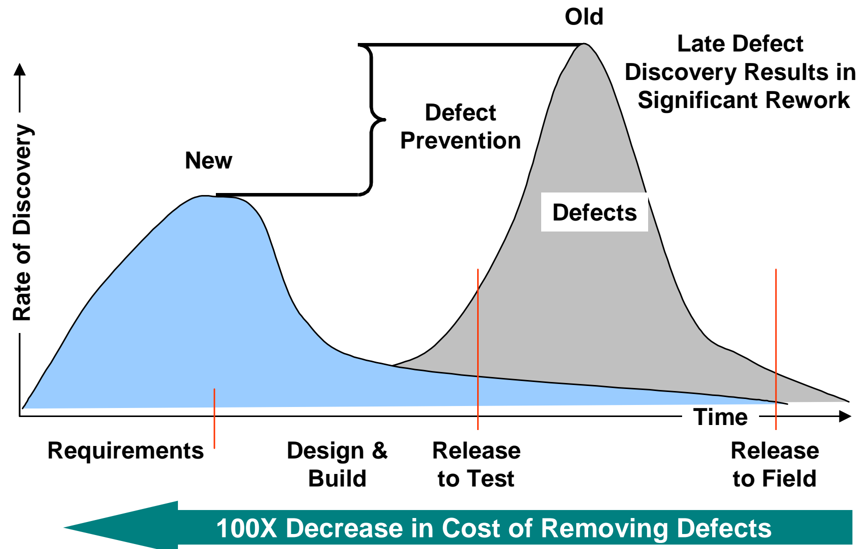



**Figure 4. Early Defect Identification and Prevention**

## 3. Defect Analysis Concepts

Requirement clarification during model development can uncover requirement problems such as ambiguities and inconsistencies. However, subtle errors or errors resulting from inherent system complexity can hide defects in a model or implementation. This section briefly describes defect types and how automated model analysis identifies them.

### 3.1 Defect Types

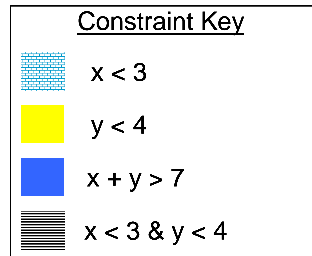
There are two types of errors: computation errors and domain errors. As defined by Howden, a **computation error** occurs when the correct path through the program is taken, but the output is incorrect due to faults in the computation along the path. A **domain error** occurs when an incorrect output is generated due to executing the wrong path through a program [How76]. Such errors can be introduced in a model as a result of errors in the requirements or during the requirement clarification process.

### 3.2 Domain Error Example

The concept of a program path and its related output computation is analogous to a requirement or design thread of a model. A domain error for a model thread means that there is no input set that satisfies the model constraints. Consider the following trivial example:

```
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
      x < 3 & y < 4
then
   maximum value for x < 3 is 2
   maximum value for y < 4 is 3
   minimum value for x + y > 7 is 8
```



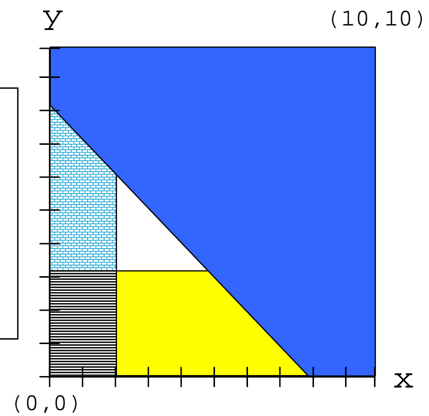




**Figure 5. Example of Inconsistent Constraints**

The region represented by the intersection of x & y does not overlap the constraint region defined by $x + y > 7$. The constraint expression is contradictory and cannot be satisfied. The contradiction results in a domain error, because the variable z will 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, subsystems or components of an application. Model problems can be hidden when constraints reference common variables that are distributed throughout several model subsystems as reflected in Figure 6. In these situations it can be difficult to isolate these types of errors through manual processes. Automated model analysis provides a tool for locating these errors.

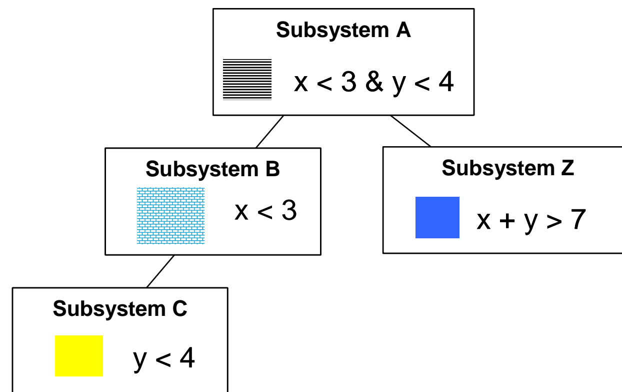



**Figure 6. Constraints Span Hierarchy of Subsystems**

### 3.3 Computational Error

Computational errors can result from various root causes such as an expression with incorrect variables, incorrect operators (+ instead of -), missing or incorrect parenthesis, or incorrect constants. Erroneous expressions can result in range errors, either underflows or overflows for the data type of the object. During test generation, low-bound and high-bound values are selected for the variables used in the computation in an attempt to stimulate range errors that can be traced to an expression with a defect. Blackburn provides examples of several computational errors that result from common errors in developing expressions for scaled arithmetic [Bla98].

## 4. Design-based Models and Verification

For design-based modeling approaches, the process resembles the illustration shown in Figure 7. Simulink/Stateflow and MATRIXx are hybrid, control system modeling and code generation tools. In this scenario, models undergo translation and static analysis to verify their integrity. The T-VEC system can identify model defects, and model checking ensures all paths through the model are valid, which means that code generated from the model is reachable. Without this capability, models can be used to generate code automatically, but the results of executing that code under certain conditions are undefined. This particular capability provides increased confidence as to the integrity of the model. Model problems are reported to the engineer responsible for constructing the model for immediate correction. Once modeling is complete, the model is used as the basis for developing tests. Through dynamic analysis (i.e., execution through auto generated code or within a simulator) of the system, anomalies in the model and implementation can be identified and corrected.

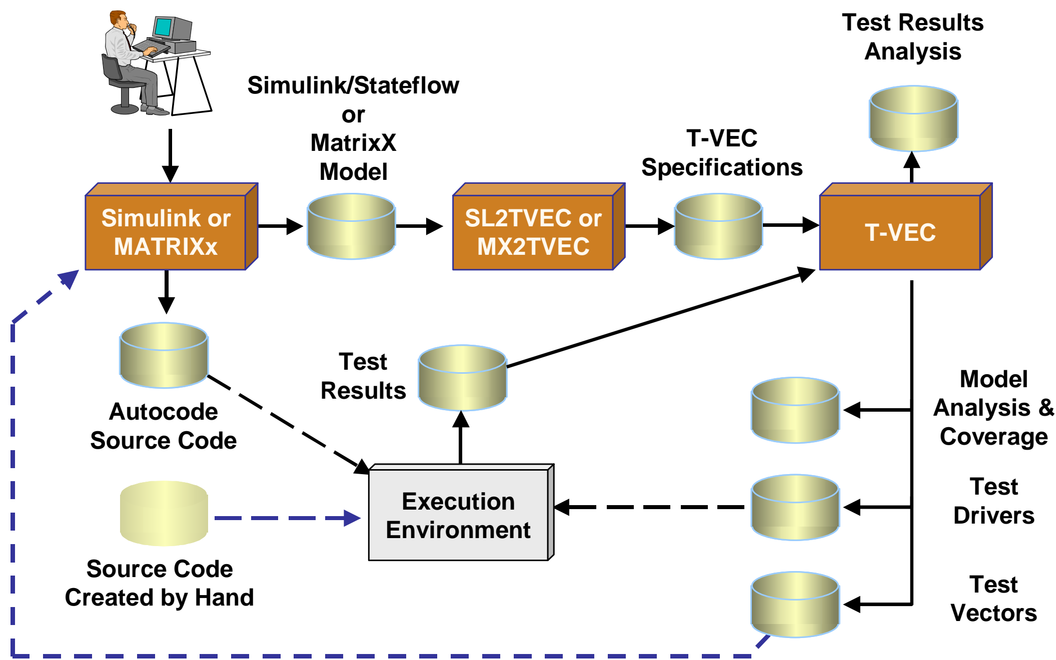



Figure 7. Simulink/Stateflow and MATRIXx Modeling Process Flow Test Sequences

Design models used for simulation and/or automatic code generation often include input-to-output relationships involving multiple cycles of execution. This is due to the use of primitive operators that have "state memory" feedback semantics in the manner of sequential logic designs described above (e.g., the TimeDelay block in Simulink). These types of operators are often used to design digital signal processing applications such as signal frequency sensitive filters and feedback-loop control law mechanisms for digital control applications. Such applications are very dependent on exhibiting a dynamic response to their input signal values.

When an application's design includes dynamic response characteristics, it is often difficult to predict the expected output value response for a given set of input values when only considering a single cycle's inputs. Consequently, verifying the correct operation of such a design is a non-trivial task and compiling verification evidence of proper functionality with traditional software testing approaches can be problematic. However, verification evidence typical of these approaches is often required by customers and certifying agencies, such as the FAA in the commercial aerospace domain.

Traditional software testing approaches are generally centered around developing and applying suites of test cases, where each test case is comprised of a set of input values and an expected output value, are geared towards verifying the required static response of a system. The system under test (SUT) is initialized with the input values, executed from a specific start point to specific end point in the application's instruction space, the actual value of one or more output variables is extracted and compared to the expected output values, and the results of these comparisons determine the pass or fail status of the test. Each such test is the examination of a single input-to-output execution cycle, essentially one state transition of the overall system. Tests of this type are expected to be repeatable any number of times in sequence – the same input values expected to result in the same output values. However, the use of operators with "state memory" can render such single state transition test cases totally non-repeatable. Each successive execution of the test can result in a unique output result. It should be apparent that such an approach to testing is inadequate, at best, for fully verifying the time-wise non-linear or state-machine-based characteristics found in such models.

It is possible to test a SUT's dynamic response using the "test case" approach by modeling "state memory" variables as additional input variables. However, it can be difficult to determine what values these "state memory" inputs should be for a given test case because they depend directly on the history of inputs. The complexity of the mechanism providing such "state-memory" semantics, and of the mathematical relationships characterizing system response in terms of inputs and this state memory, is primarily responsible for this difficulty.

The requirements governing dynamic response are often expressed in terms of output value tendencies, such as *rise time*, *over shoot*, and *settling time* rather than functional value mappings between a single input value set and an associated output value.

Requirements describing a system's static response can be formally expressed in terms of pre-condition/post-condition pairs. The pre-condition characterizes the system states under which the post-condition's input-values-to-output-value mapping is required to hold. The requirements governing a given output can be said to be "complete" if there is at least one pre-condition/post-condition pair describing the value of the output in terms of input values for all points in time for all modes of operation of the system. They can be said to be "consistent" if there is at most only one such pre-condition/post-condition pair for a given output variable for any given point in time.

A set of test cases is associated with a complete and consistent set of pre-condition/post-condition pairs that can be shown to produce MC/DC-complete requirements-based tests. A suite of such test cases, when used to drive an implementation intended to satisfy these requirements, provides sufficient evidence that the implementation does indeed effectively satisfy them, at least from a functional point of view. The T-VEC system has demonstrated that the automatic generation of a set of such tests can be accomplished.

### 4.1 Testing a Model With Feedback Semantics

An example of a model that employs both time-wise non-linear computational feedback elements as well as state-machine-like elements is the Flow Control model shown in Figures 1, 2, and 3.

The Flow Control Model design employs a simple first-order lag filter (temperatureSensor subsystem), applied to the temperature input data signal *In1,* and a small "hysteresis" based threshold detection state machine (flowControlLogic subsystem). Each of the two primary subsystems includes a TimeDelay primitive operator block. This operator is used to retain the value of an intermediate computation result from one cycle of execution and provide that same value as an input to the next cycle's computation. The TimeDelay block provides a generic closed-loop feedback mechanism useful for constructing simple state

machines and also for implementing digital signal processing algorithms such as filters and digital control law algorithms.

The required operation of the Flow Control model is the following:

1. The flowControlLogic state machine (Figure 9) is required to output the value of 0 during the current cycle if it had output a 0 during the previous cycle and the value being output from the temperatureSensor subsystem during the current cycle is less than or equal to 180 degrees. When flowControlLogic outputs a 0 during the current cycle, the flowControl system should also output the value of 0, regardless of the specific value being input to and output from the temperatureSensor subsystem.
2. The flowControlLogic state machine is required to output the value of 1 during the current cycle if the value output from the temperatureSensor subsystem during the current cycle is greater than 180 degrees, no matter what value it output during the previous cycle. While flowControlLogic outputs the value of 1, the main flowControl system is required to output a value based on the value produced by temperatureSensor, after being scaled through addition and multiplication operations.

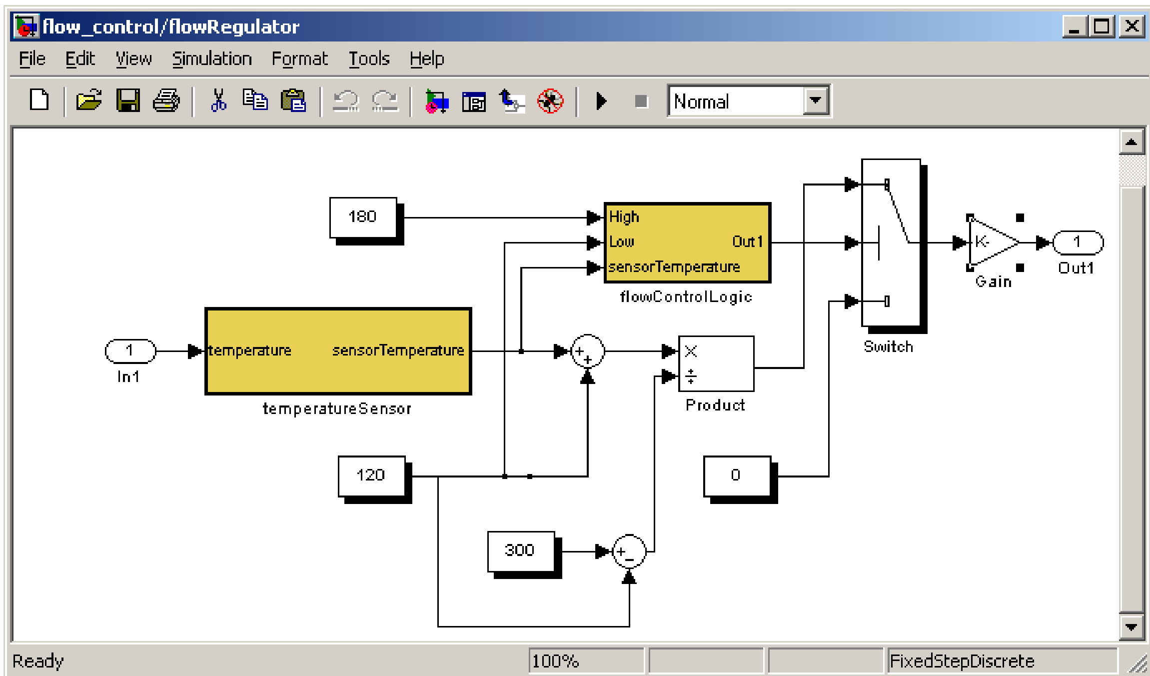



**Figure 8 - Flow Control Model**

3. The flowControlLogic state machine is required to output the value of 1 during the current cycle if its previous cycle output was 1 and the value being output from the temperatureSensor subsystem during the current cycle is greater than or equal to 120 degrees. While flowControlLogic outputs the value of 1, the main flowControl system is required to output a value based on the value being output by temperatureSensor after being scaled through addition and multiplication operations.
4. The flowControlLogic state machine is required to output the value of 0 during the current cycle if its previous cycle output was 1 and the current value being output from the temperatureSensor subsystem during the current cycle is below 120 degrees. This results in the main flowControl

system outputting the value of 0 during the current cycle, regardless of the specific value being output by temperatureSensor.

5. The temperatureSensor subsystem block (Figure 10) is required to provide simple first order filtering. If the filtered value of temperature is between the saturation limits of –100.0 to 300.0 degrees, the output is required to be equal to a "filtered" temperature value. This "filtering" results in an averaging effect, preventing spurious "noise" spikes in the value of temperature from being passed through to the flowControlLogic state machine and thus causing it to trigger an undesired state change. This effect can be seen in a graph of the dynamic input response to a standard step input signal in Figure 11.
6. The temperatureSensor subsystem block is required to saturate at low bound and high bound value limits. If the filtered value of the temperature signal input is below –100.0 degrees, temperatureSensor will output –100.0 degrees (6a). If the filtered value of the temperature signal input is above 300.0 degrees, temperatureSensor will output 300.0 degrees (6b). (Note – in the case of the overall Flow Control model (Figure 8), the flowControlLogic state machine will prevent any value of filtered temperature below 120 degrees from ever being output from the system.)

From this description of the required operational semantics of the Flow Control model, it should be clear that the traditional black-box testing approach that sets input values, executes the code through one execution cycle, extracting output values, and comparing the results, would be inadequate. For example, the dynamic response curve of Figure 11 clearly indicates that it takes nearly 0.4 of a second (with a sample period of 0.1 seconds) for the output of temperatureSensor to rise to from 0.0 to its full value of 100.0 degrees in response to a step input signal of 100 degrees that takes place at t=0.0 in the simulation run.

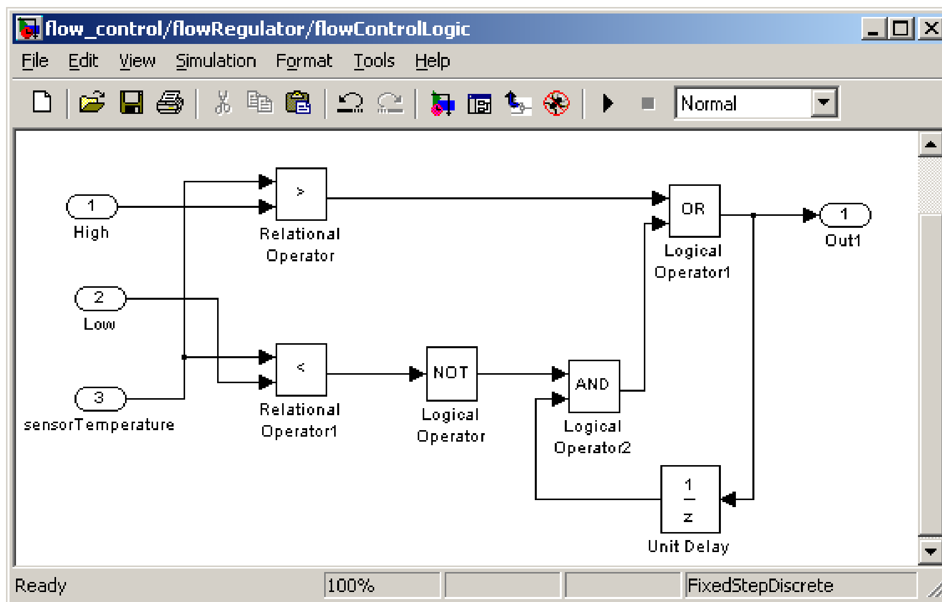



**Figure 9 – FlowControlLogic State Machine**

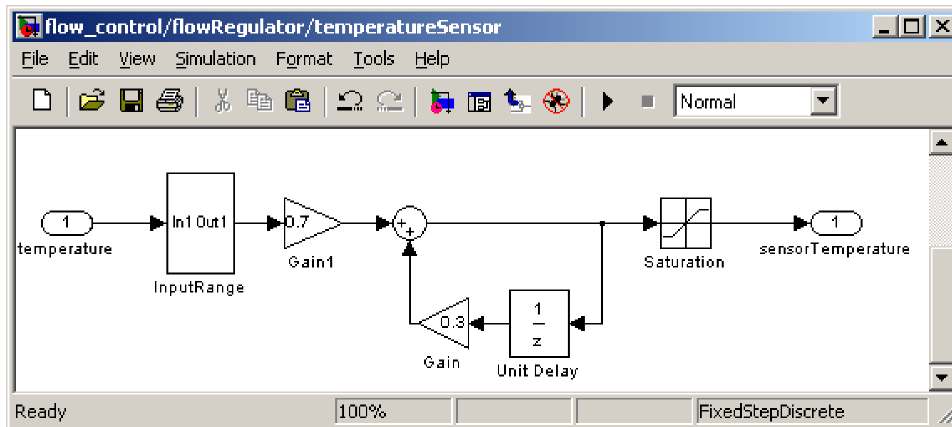



**Figure 10 - First Order Filter**

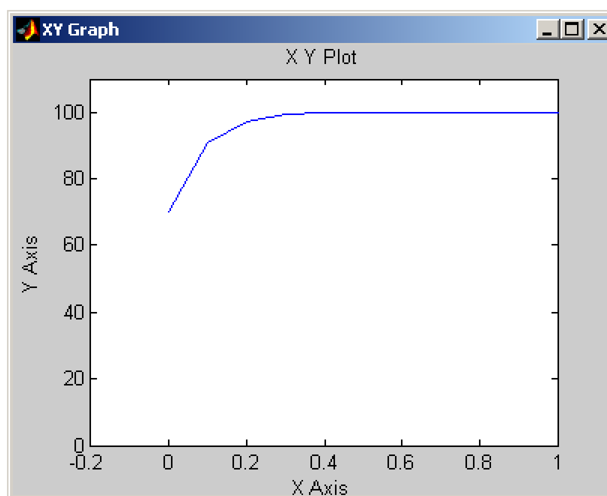



**Figure 11 - temperatureSensor Response to Step Input of 100 Degrees**

To verify that a given implementation of this model correctly provides such a response to a step input signal, one would need to test the implementation's response over a period of time, (i.e., numerous cycles of execution). Consequently, test cases that include an association between a single set of input values and a single expected output value cannot adequately verify such performance. What is required is a new concept in specification-based software test generation, test sequence vectors (TSVs).

Informally, a TSV is a test specification that includes all of the input values for a sequence of execution cycles (i.e., invocations) of the system being tested. A TSV includes values for each independent input variable (e.g. *temperature,* in the Flow Control model) for each invocation of the model. It also contains initial condition values for the closed-loop feedback variables used by the first invocation in the sequence. A TSV includes expected output values for each individual system invocation in the sequence, as well as the final expected output values for the overall sequence. A TSV for a 4-step sequence of the flowRegulator model is conceptually depicted by Figure 12. This represents 4 sample periods of execution of the cyclic flowRegulator model. Sequence values can be specified explicitly or using functions, such as step, ramp and impulse.

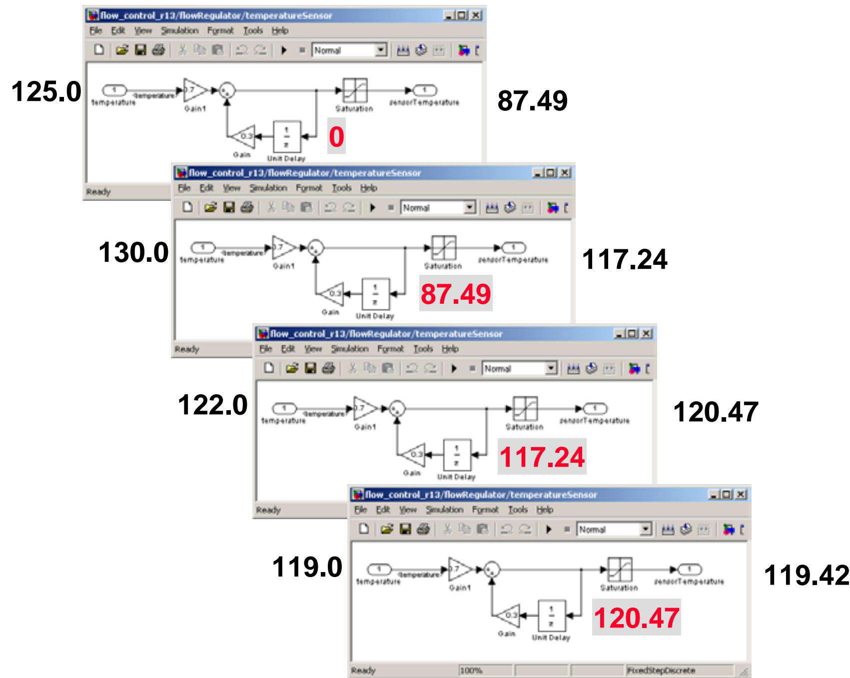



**Figure 12. Sequences Includes Feedback of Unit Delay**

## 5. Process for High Integrity Systems

Depending on the software level of the system being considered for certification, the decision flow shown in Figure 13 may be required to provide evidence that the model is defect free and that the generated tests provide the required level of code coverage. Details associated with several of these steps are provided below. The process is a follows:

- Construct a model in Simulink, MATRIXx, or TTM.
- Check the model for defects and iteratively correct the model if there are defects.
- Construct the code. This can be a manual process or can be supported using automatic code generation capabilities supported by tools like Simulink and MATRIXx.
- Generate the tests.
- Execute the tests through instrumented code.
- Check to ensure that the tests provide adequate coverage (e.g., MC/DC coverage); if adequate coverage is not achieved, additional tests must be generated.
- Check to ensure that all tests pass.
- Execute tests against target code.
- Check to ensure that all tests pass.
- If tests do not pass, perform test failure analysis, and correct the code or model.

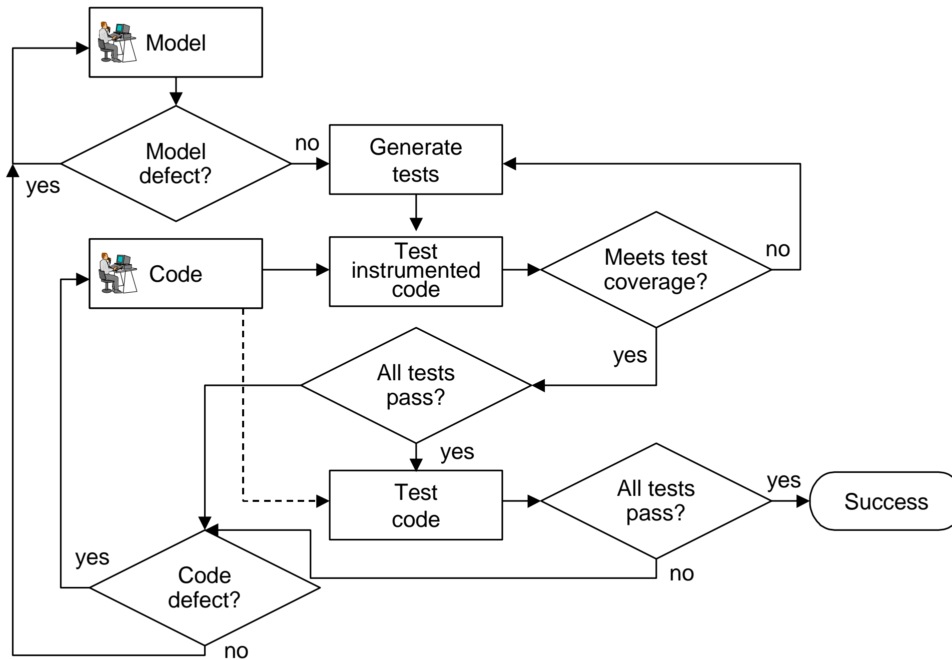



**Figure 13. Verification Decision Flow**

## 6. Summary

This paper describes how the model-based Test Automation Framework (TAF) for software verification supports early defect identification through model checking and automatic generation of tests from requirement-based or design-based models. Using the TAF model translators, requirement-based or design-based models are converted into a form where T-VEC, the test generation component of TAF, produces tests vectors that include inputs as well as the expected outputs, and test drivers to execute tests in various environments. TAF is integrated with requirement management tools, such as DOORS, to provide full traceability from a requirement to a generated test case. TAF is also integrated with code coverage-based tools that allow the generated tests to be measured for code-based tests coverage. Such an approach reduces requirement defects, manual test development effort, rework, and helps reduce the cost of developing and verifying high integrity software intensive systems.

A growing number of high integrity systems are being developed using model-based system with automatic code generation such as Simulink and MATRIXx. The cost of verification imposed by certification authorities such as the FAA has costly implication. This paper also recommended a process to use qualified model-based tools to automatically analyze and generate test vectors to systematically provide evidence that meets the highest FAA standards for certification of these systems. Most importantly, development teams have reported that these capabilities help address the significant needs for verifying high integrity systems that continue to increase in complexity, and have helped companies reduce the cost of such efforts by 50% over the traditional manually intensive processes.

## 7. References


[AFBPP92] Alspaugh, T.A., S.R. Faulk, K.H. Britton, R.A. Parker, D.L. Parnas, and J.E. Shore. *Software requirements for the A-7E aircraft*, Tech. Rep. NRL/FR/5546-92-9194. Washington, D.C.: Naval Research Lab, 1992.

[Bla98] Blackburn, M. R., Using Models For Test Generation And Analysis, Digital Avionics System Conference, October, 1998.

[BBN03] Blackburn, M.R., R.D. Busser, A.M., Nauman, Interface-Driven, Model-Based Test Automation, CrossTalk, The Journal of Defense Software Engineering, May 2003.

[DOT99] U.S. Department Of Transportation, Federal Aviation Administration, Order 8110.83 -*Guidelines For The Qualification Of Software Tools Using RTCA/DO-178B*, April, 1999.

[HJL96] Heitmeyer, C., R. Jeffords, B. Labaw, Automated Consistency Checking of Requirements Specifications. ACM TOSEM, 5(3):231-261, 1996.

[How76] Howden, W.E., Reliability of the Path Analysis Testing Strategy, IEEE Transactions on Software Engineering, 2(9):208-215, 1976.

[BB96] Blackburn, M.R., R.D. Busser, T-VEC: A Tool for Developing Critical System. In Proceeding of the Eleventh International Conference on Computer Assurance, June, 1996.

[Mil98] Miller, S. P., 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.

[NCS99] Nelson, M.T., J. Clark, M. A. Spurlock. "Curing the Software Requirements And Cost Estimating Blues," PM: Nov-Dec, 1999.

[Off99] Offutt, A.J., Generating Test Data From Requirements/Specifications: Phase III Final Report, George Mason University, November 24, 1999.

[Sta99] Statezni, David, Industrial Application of Model-Based Testing, 16th International Conference and Exposition on Testing Computer Software, June 1999.

[Sta00] Statezni, David. Test Automation Framework, State-based and Signal Flow Examples, Twelfth Annual Software Technology Conference, May 2000.

[Saf00] Safford, Ed, L. Test Automation Framework, State-based and Signal Flow Examples, Twelfth Annual Software Technology Conference, May 2000.

[WIK92] Wallace, D.R., L.M. Ippolito, and D.R. Kuhn, High Integrity Software Standards and Guidelines, NIST SP 500-204, National Institute of Standards and Technology, Gaithersburg, MD, 20899, September, 1992.

[WJ91] Weyuker, E., B. Jeng, Analyzing Partition Testing Strategies, IEEE Transactions on Software Engineering, 17(7):703-711, 1991.

[WC80] White, L.J., E.I. Cohen, A Domain Strategy for Computer Program Testing. IEEE Transactions on Software Engineering, 6(3):247-257,May, 1980.

[Zei89] Zeil, S.J., Perturbation Techniques for Detecting Domain Errors, IEEE Transactions on Software Engineering, 15(6):737-746, 1989.