ソフトウェアテストシンポジウム 2005
JaSST'05: Japan Symposium on Software Testing 2005

制御系システムモデルからのテストシーケンス生成

Test Sequences Generation from Control System Models

Mark Blackburn Ph.D., Robert Busser, Aaron Nauman, Travis Morgan
T-VEC Technologies, Inc.　　2214 Rock Hill Road, Herndon, VA 20170　USA
（国内連絡先：富士設備工業㈱浅野義雄　yoshio@fuji-setsu.co.jp ）

あらまし　制御系システムの動的な状態に対するテストシーケンスを制御系デザインモデル Simulink® などから生成するテストベクタ生成システムについての最新の成果を報告します。制御系システムで一般的なタイムディレイやインテグレータなどのフィードバックループを用いたモデルをサポートします。テストシーケンスは論理パスを解釈し、システムの動的な状態を含めてテストベクタを算出します。またこの論文では、モデルベーステストツールを使用し検証のエビデンスを生成するプロセス(FAA の DO-178B などに適応する)に関してや、それに対するツールとしてのクオリフィケーション対応についても触れています。

Abstract　This paper discusses some recent advancements of the test generation system to produce test sequences for testing dynamic systems from design modeling systems such as Mathworks' Simulink®. Test sequences support test generation of systems that are modeled using constructs that support feedback, such as integrators or time delays, which are common in control system models. Test sequences can address the logic paths, and computation testing in the software as well as dynamic aspects of systems response. The paper briefly discusses tool qualification support, and processes for using this model-based testing tool to produce verification evidence that meets the FAA standards such as DO-178B

1. Introduction

Requirement and design-based models are used in aircraft and automotive software-system development where high reliability is demanded. Some 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 [1].

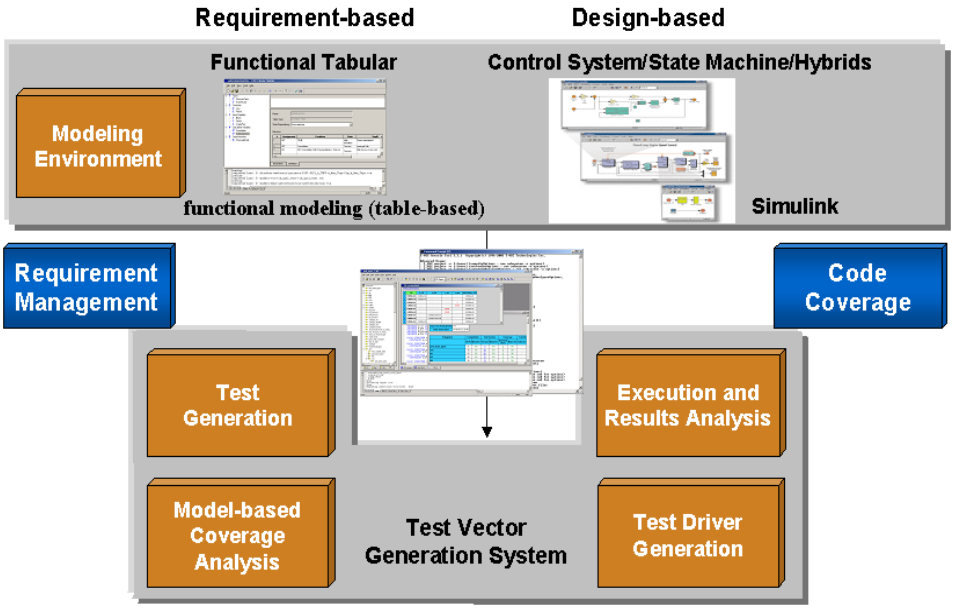

Figure 1. TAF Integrated Components

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 model analysis and test generation for requirement-based tools which is a functional (table-based) modeling tool based on the Software Cost Reduction (SCR) method [2]. TAF also supports model analysis and test generation for design-based modeling, simulation, and code-generation tools such as MathWorks' Simulink and Stateflow.

Through the use of model translation, requirement-based or design-based models are converted into a form where 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. Test vector generation system 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, which produces test drivers that are then executed against the implemented system during test execution.

1.1 Usage Scenario

For design-based modeling approaches, the process resembles the illustration shown in Figure 2. Simulink/Stateflow is a hybrid, control system modeling and code generation tools. In this scenario, models undergo translation and static analysis to verify their integrity. The Test vector generation system is its ability to identify model defects. The 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.

1.2 Background

The core capabilities of this approach were developed in the late 1980s and proven through use in support of FAA certifications for flight critical avionics systems [3]. The approach supports requirement-based test coverage mandated by the FAA with significant life cycle cost savings [4; 5; 6].

The approach 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 a most languages (e.g., C, C++, Java, Ada, Perl, PL/I, SQL), as well as, proprietary languages, and test environments. The TAF 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, 8110.49, Chapter 9, Qualification Of Software Tools Using RTCA/DO-178B [7].

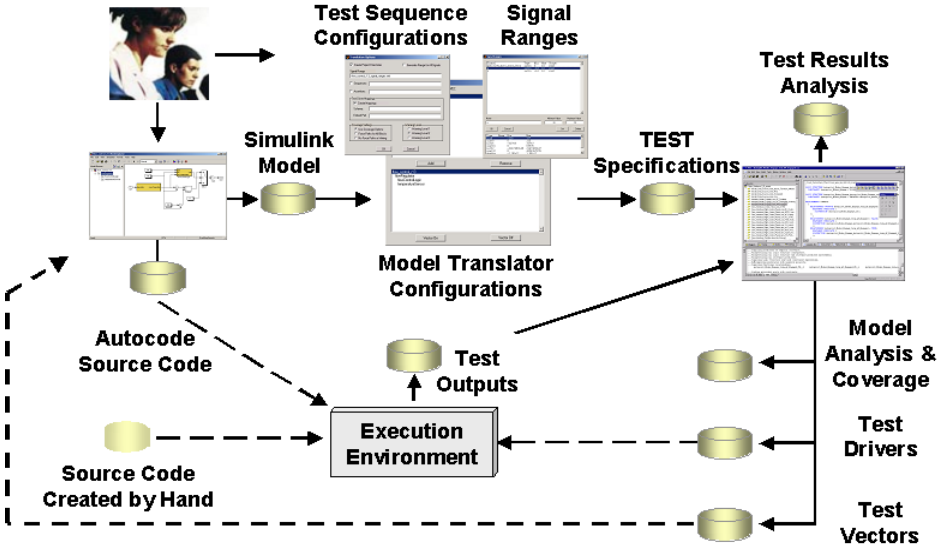

Figure 2. Simulink/Stateflow Modeling Process Flow

1.3 Scope

This paper focuses on a test generation system to produce test sequences to support the testing of dynamic systems. While TAF/Test vector generation system has been used primarily for verifying the static response of systems under test, this paper describes how this approach can be applied to the generation of test cases suitable for verifying the performance of systems that also exhibit dynamic response. It describes how the Test vector generation system mechanism can be configured to include the semantics associated with multiple cycles of "execution" of a given system. This description shows how Test vector generation system can be used to automatically determine input values for the "state memory" variables that characterize such systems. In addition, it shows how the powerful state-space solver capabilities of this technology can be employed to assist in synthesizing many of the important gain coefficients such designs depend on.

1.4 Concepts and Definitions

Logic design is often classified into two categories, "combinatorial" and "sequential". Combinatorial logic is comprised strictly of stateless logic operators (AND, OR, NOT, etc.). The output values produced by combinatorial logic formulae are expected to be exactly the same for the same set of inputs values. Systems based on combinatorial logic maintain no memory of input values or of computation results from previous execution cycles and can be said to exhibit a static response to the values of its input variables. The output response of the system never varies for a given input excitation with respect to the current execution cycle. Sequential logic includes one or more components whose purpose is to maintain some knowledge of the input values from previous execution cycles, such as flip-flops or time-delay operators, and whose output values depend not only on the current inputs but also on this historically maintained knowledge. The output response of a sequential logic system can vary from cycle to cycle in reaction to the history of inputs experienced by the system. Systems of this type can be described as exhibiting a dynamic response in addition to any static response characteristics they may have.

2. Overview

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, is 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" semantics 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 MCDC-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 satisfies 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.

3. 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 employees 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 4) 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.

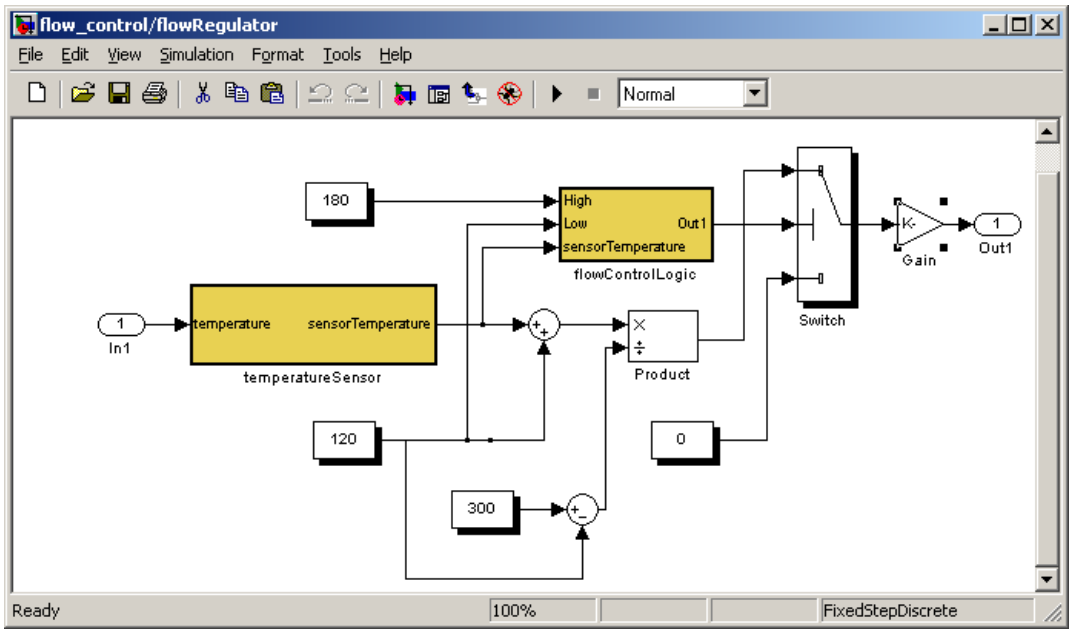

Figure 3 - Flow Control Model

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 5) 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 6.
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 3), 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 6 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.

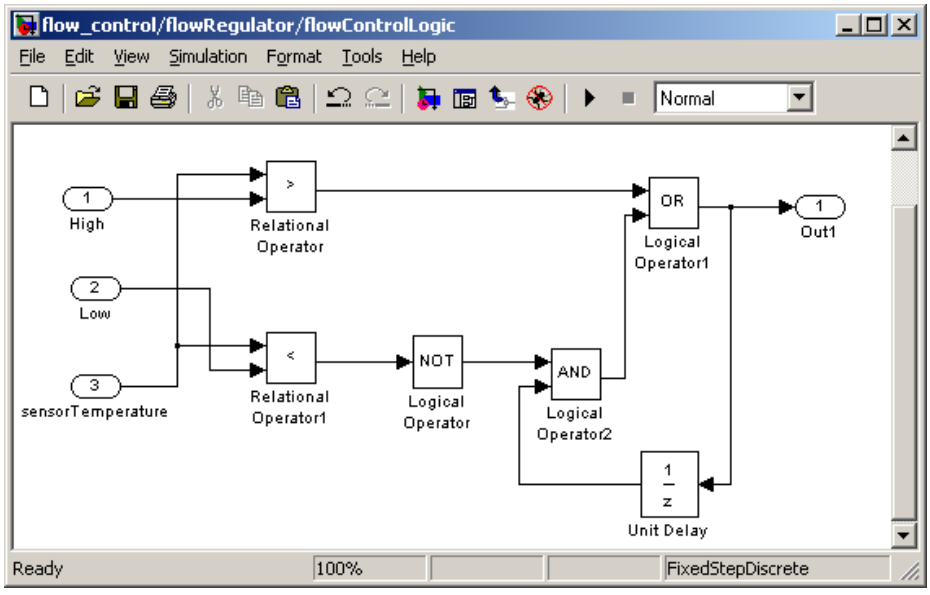

Figure 4 – FlowControlLogic State Machine

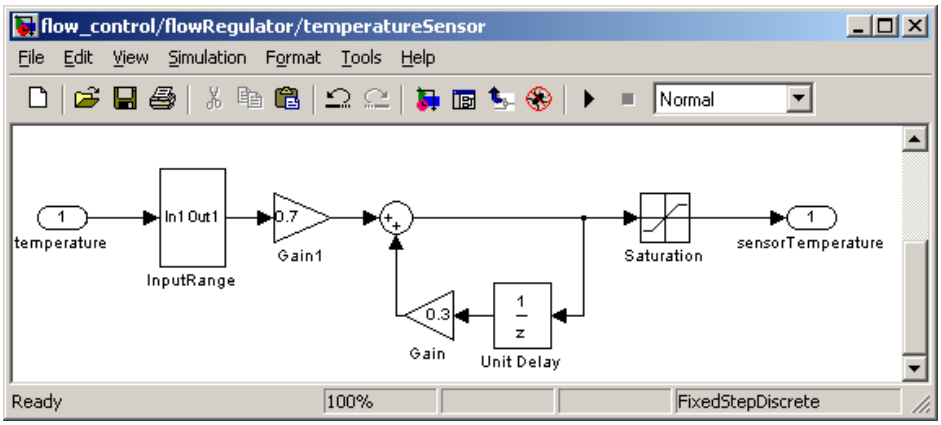

Figure 5 - First Order Filter

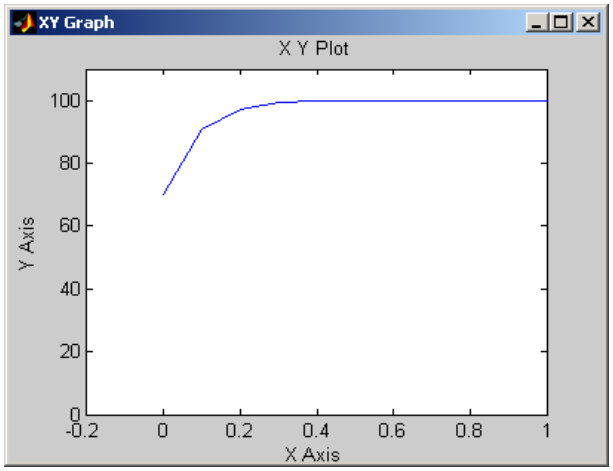

Figure 6 - temperatureSensor Response to Step Input of 100 Degrees

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 utilized by the first invocation in the sequence. Lastly, 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.

4. Summary

This paper has briefly describes some capabilities of the TAF that support test sequence generation for verification and analysis of dynamic systems modeled in tools such as Mathworks' Simulink. A TSV generation for a 4-step sequence invocation of the flowRegulator model is conceptually depicted by Figure 7. This represents 4 sample periods of execution of the cyclic flowRegulator model. The presentation will discuss test sequence vector generation using an example to illustrate both the functionality of the simple state machine in flowControlLogic as well as the signal filtering action of the temperatureSensor subsystem.

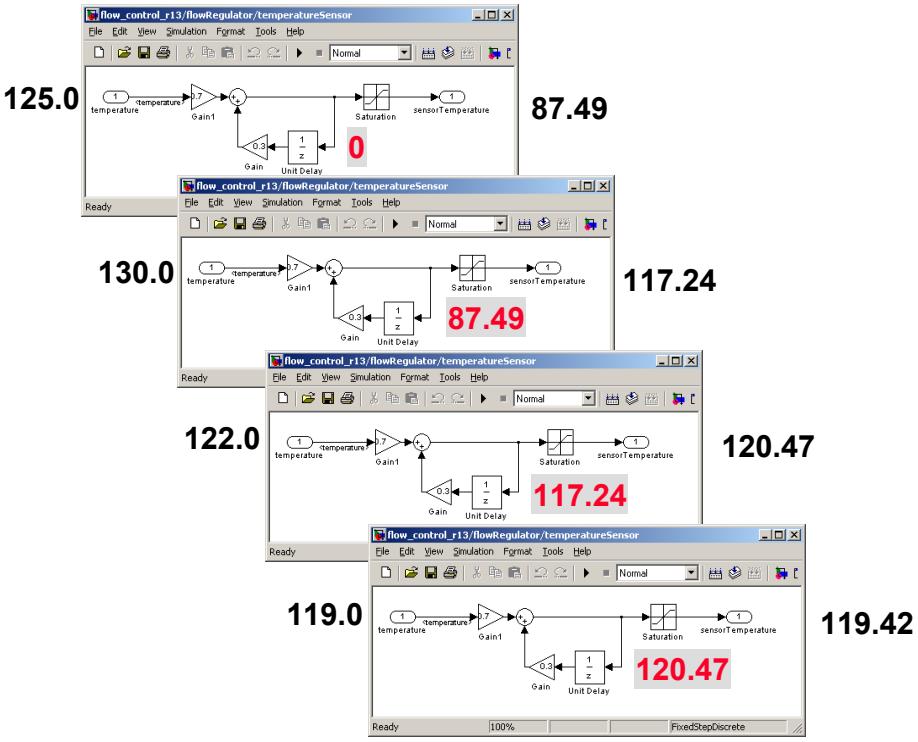

Figure 7 – Sequences Includes Feedback of Unit Delay.

5. Process for High Integrity Systems

Depending on the software level of the system being considered for certification, the decision flow shown in Figure 8 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.
- Check model for defects and iteratively correct the model if there are defects.
- Construct the code. This can be a manual process or supported using automatic code generation capabilities supported by tools like Simulink.
- 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.

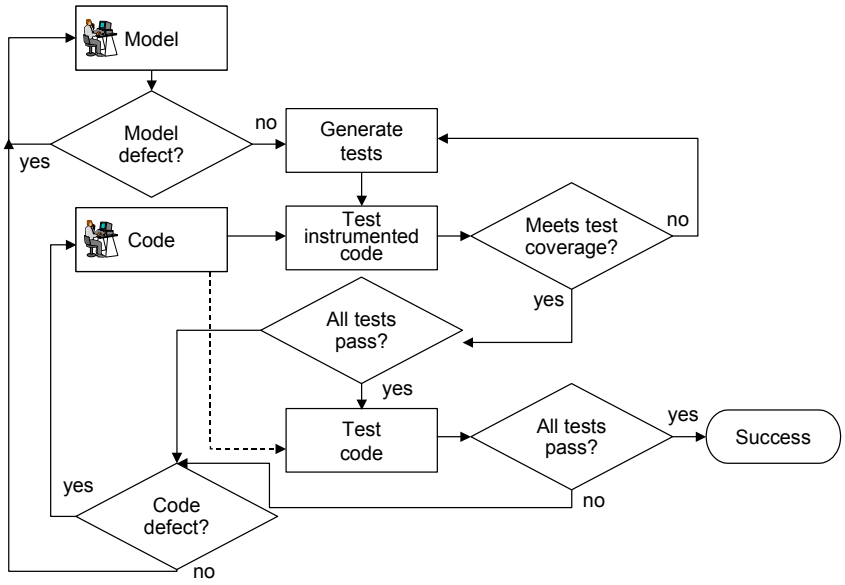

Figure 8. Verification Decision Flow

6. References

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

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

[3] 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.

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

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

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

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