A Benchmark Problem for Model Based Control System Tests

Yogananda Jeppu (yvj_2000@yahoo.com),
Chethan CU (chethan.cu@gmail.com),
Prasad K (k14prasad@gmail.com),
Selvamurugan Hariram (hariramsatheesh@gmail.com),

Moog India Technology Centre
Bangalore, India

Prakash R Apte (apte@ee.iitb.ac.in)
Professor, Indian Institute of Technology
Mumbai, India

Background

Today safety critical flight control systems are tested using model based approach. The model blocks are proprietary and seldom shared in the open. A benchmark problem was designed as part of a research activity to test out certain test case generation techniques. This model was also used as a problem for the test case generation methodology training classes. Trainees, normally fresh graduates from colleges, were asked to design manual test cases to find out the errors embedded in the model. The control system blocks are typical of the ones used in a flight control system or an automobile control system. It is a combination of linear filters, integrators, non-linear blocks like rate limiters and lookup table. There is a combination of logic and time dependency in terms of persistence blocks. These are however more complex for generating test cases as explained below. The blocks have been selected and placed to ensure that the students exercise some thought process and understand the underlying functionality of the control system blocks.

This problem is being provided as open source to the Control System test community. This we believe is a first of its kind and we hope to provide more benchmark problems as we go along comparing Taguchi method of testing. Users are free to try out the Matlab or other commercial test tools against this model. We would like to hear about your experience in using this benchmark problem.

Benchmark Problem

The benchmark model is called compete_2010.mdl. This is a Simulink model with a test harness, which takes in 10 inputs from the workspace named *Inp1* to *Inp10*. It has 7 outputs, which come out as a vector in a variable *simout* (refer Figure 1). There is a Matlab code, which is an exact replica of the model in code form (file model00001.m). This has undergone more than 20,000 test runs and both model and code match very well. There is a variation of this with instrumentation for coverage called model00001_c.m. This file has a variable COVERAGE, which collects the line, condition and logic coverage. There is a Simulink model with 17 mutants injected into the model subsystems

called compete.mdl. This has the model and the mutants in the same file. Each mutant block has a single mutant embedded into it. The output of the mutant block and the original model is compared and is available as error in the scope. The model output is Output in workspace and the mutant outputs are available as Output1 to Output17 in the workspace.

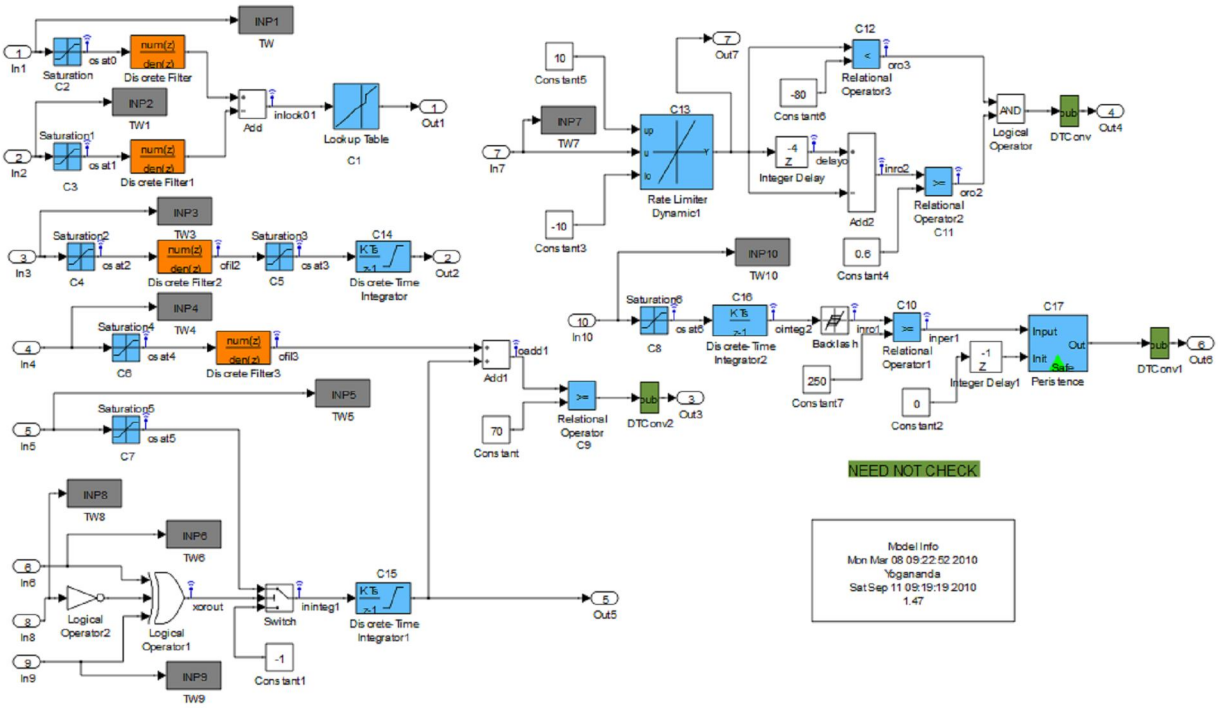

Figure 1. The complete model

The model is composed of 5 separate models. This has been done to confuse the test designers to think that it is a 10 input problem. Model 1 is shown in Figure 2. This is a two input block with both the inputs limited to ± 60.0. The limited signal passes through two-second order filters with a low damping coefficient. This was deliberately chosen so that the output of the filter would have a large overshoot. The output of the two filters are subtracted and injected into a table lookup block. The subtraction is carried out to ensure that the team design test cases with opposite signs to ensure that the addition block output is maximized and minimized. The table lookup block is a linear interpolator block with X values chosen to have values less than -120 and greater than +120. The 120 limits are due to the saturation blocks at the input. The idea behind this selection was to ensure that the team thought beyond the limits and found ways to excite the lookup table even though the limits were placed on the inputs. This would also emphasize the concept that it is very likely that the system can have a larger value inside due to the dynamic nature of the control system blocks. Figure 3 shows a dynamic simulation of Model 1. The large overshoots of opposite signs ensure the table coverage.

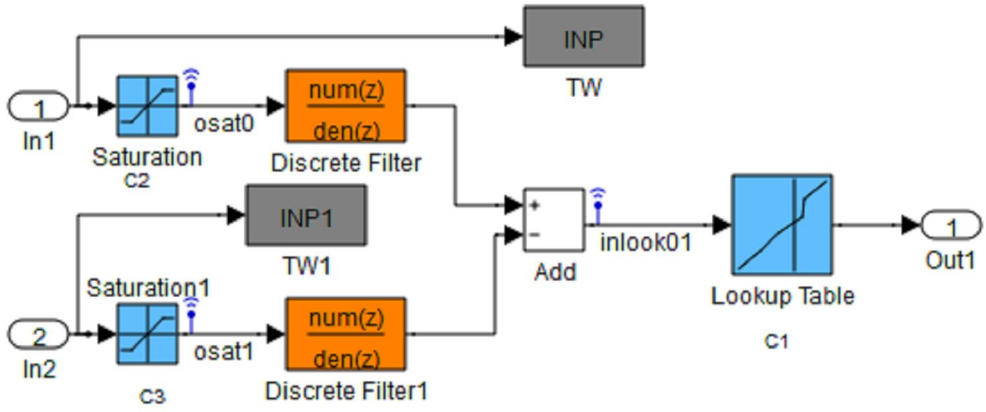

Figure 2. Model Component 1

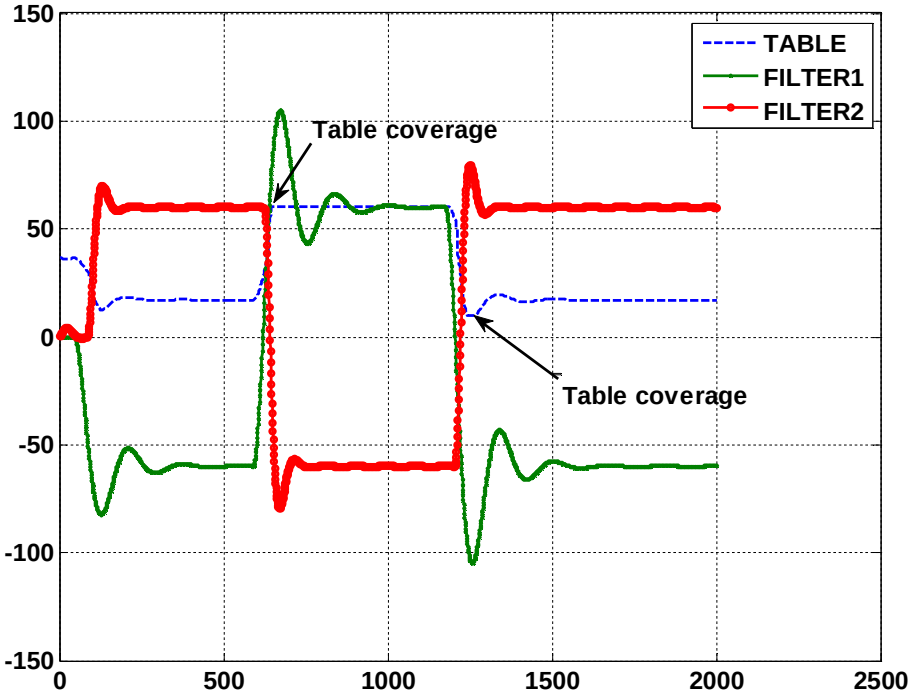

Figure 3. Filter output and the lookup table coverage

The second model has a rate limiter with an interesting computation. This is a 4-frame delay block whose output is subtracted from the current rate limited signal. This absolute error should be less than equal to 0.6 to have a True signal. This goes as an input to an AND block. The other input of the AND block is comparison with -80. The output should

be less than -80 to have a True output. The idea behind this set-up is to emphasize to the team the difference between a rate limiter and a saturation limit or an amplitude limiter. The test case designer can easily inject a large doublet to make the input do rate saturation. The rate limited output will ramp with a slope of 10 units/sec or with a sampling of 0.02 seconds it comes to 0.2 units/frame. Thus a 4-sample delay will cause the difference to be equal to 0.8. The trick is to design an input, which does not hit the rate limit. The input rate has to be brought down to below 0.6, which can be done by having a slow ramp input being injected into the rate limiter. Another thing a tester has to take care is to see that the effect is observed at the output of the system. This can be done only if the input signal amplitude is less than -80. Thus making the second input to the AND gate True. An easy way to do so is to hold the signal below -80 for a long time so that the output reaches this value and later hold constant driving the rate to 0.0.

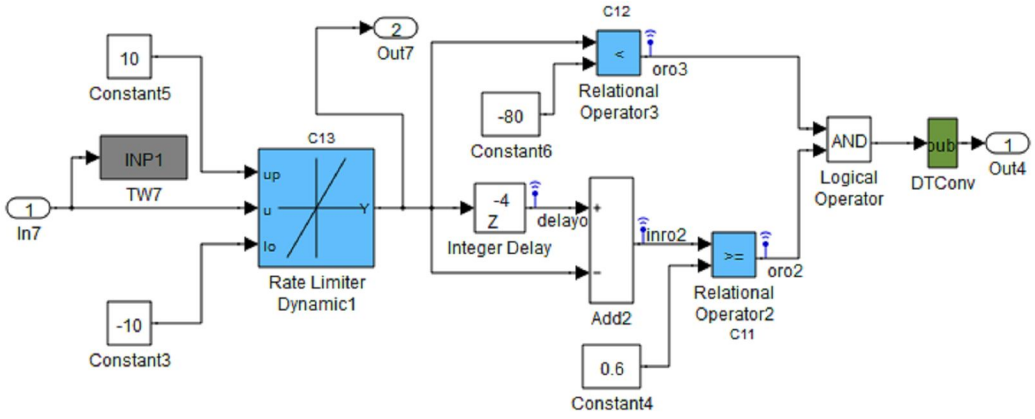

Figure 4. Model Component 2

The third model (Fig 5) is a saturation, which limits the inputs between 10.0 and –20.0. This limited signal is fed into a differentiating filter 10s/(s+10). The output of the filter is again limited to 5 and –5 and injected into an integrator. The intent using a differentiating filter is to make the team think out of using static tests with constant variables. Constant values will not excite the integrator, as the differentiating filter will drive such inputs to 0. A saw tooth waveform is ideal for this situation, which slowly charges the integrator like a capacitor. This waveform is shown in Figure 6.

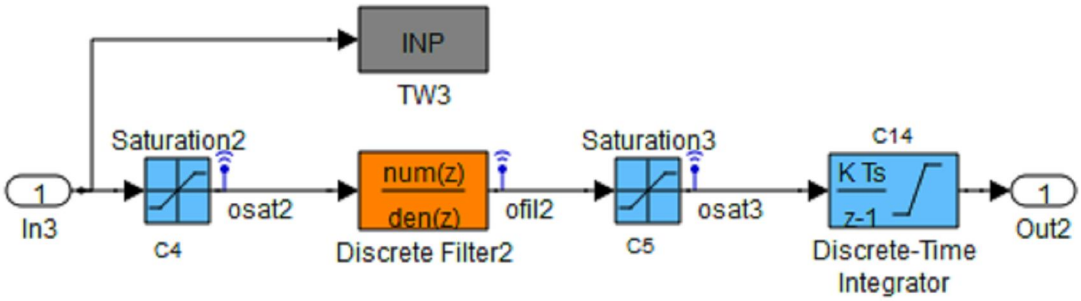

Figure 5. Model Component 3

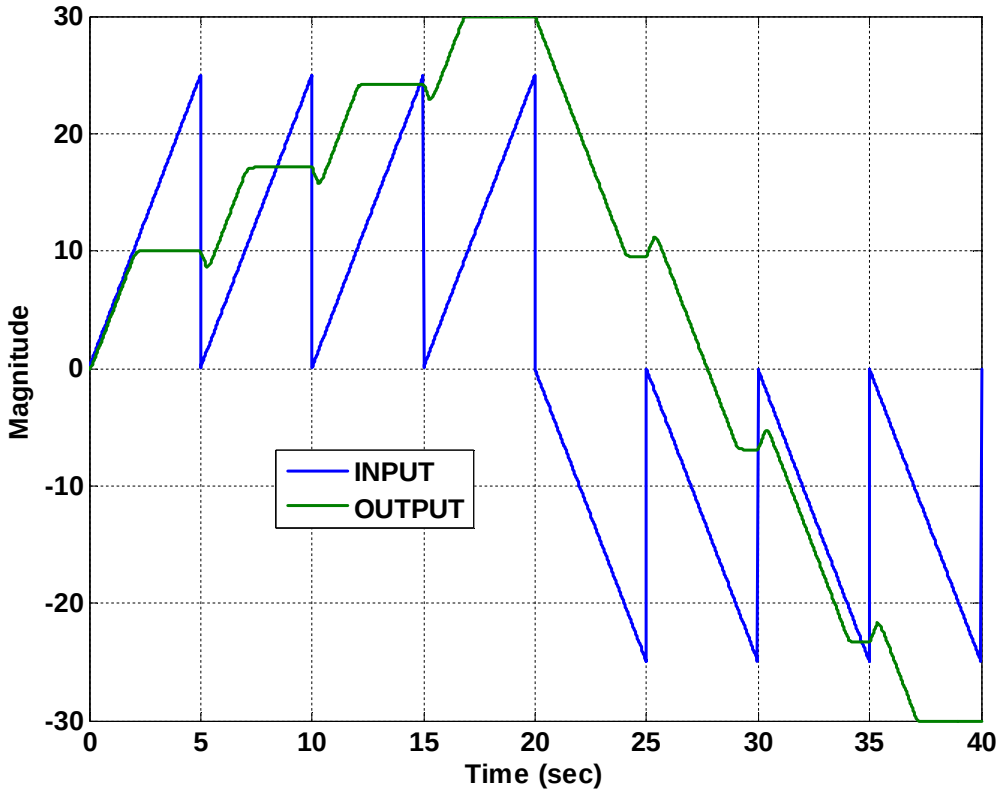

Figure 6. Integrator input and output

The fourth model (Fig 7) has a comparator that has to be tested by a set of 5 inputs. The problem has been designed to ensure that the team thinks long duration tests and combines the 5 inputs in such a way that the blocks are excited to their limit and show their functionality. The second order filter again has a low damping with the input limited to 40 and –40. The integrator output is saturated to 10 and –10. In a normal case the maximum output achievable at the comparator input is 50. A sudden step response can

drive this input beyond 70.0. The tester has to ensure that the integrator is saturated when this step input is given to the filter. The integrator takes some time to saturate, as the input to the integrator is limited to 3.5 and –3.5. The logic has to be set to True during this operation, as a False will bring it out of saturation, as the switch will drive the integrator with –1.0.

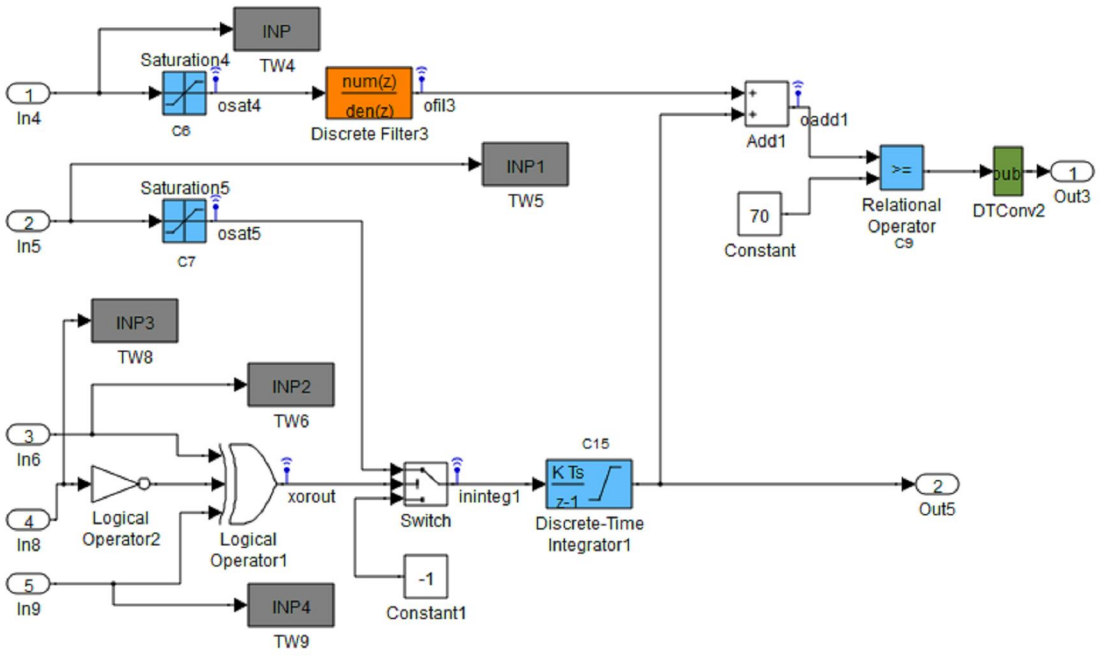

Figure 7. Model Component 4

Figure 8 has an interesting test set-up with a backlash. The backlash is excited by the output of an integrator block. The integrator output and the increment at each time frame are dependent on the amplitude of the input. This is explained by an example. Let us say the input to the integrator is 1.0. The sampling time is 20 msec for the system. The output of the integrator will then increment in steps of 1.0*0.02 or 0.02 units. An input of 0.1 at the integrator will show an increment step of 0.002. This step size indicates the capability of the signal to find errors in the backlash block or in other words the input sensitivity. Smaller the increment the smaller the error it can find out. This block emphasises the testers thought process to test with large signals and test with very small signals also. The tester will have to use a large signal to ramp up fast to a value of 250 to test the relational operator and he or she should have a small enough signal also to test the backlash.

The complexity of the model is further increased by the addition of the persistence block. This blocks checks to see if the input signal was true for a specified duration (in this case 100 frames) to declare a True output. It also checks for an input false for 150 frames before declaring a false. Such block combinations are extensively used in safety critical control signals to vote out a bad signal or declare a signal healthy. The tester will have to

play around with the integrator input to ramp up 250 and beyond to test the persistence for True. Then he/she has to set the output to less than 250 to check the persistence for False. Figure 9 shows the persistence output.

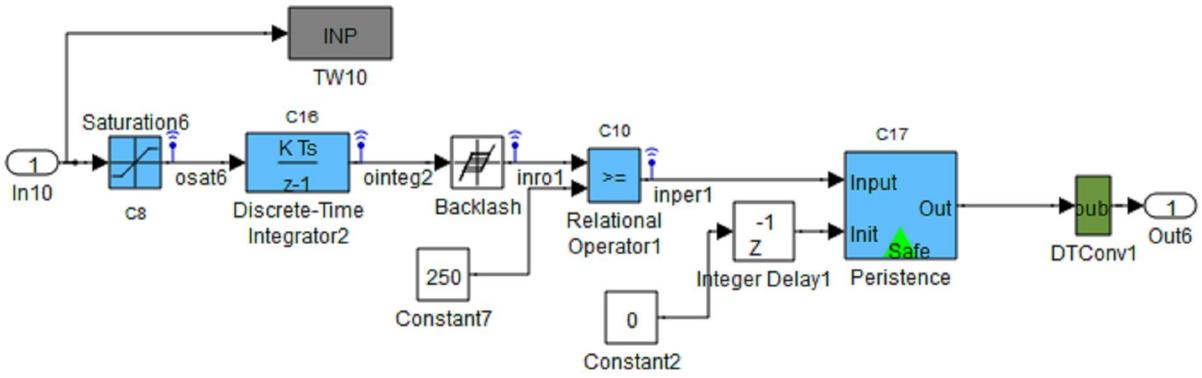

Figure 8. Model Component 5

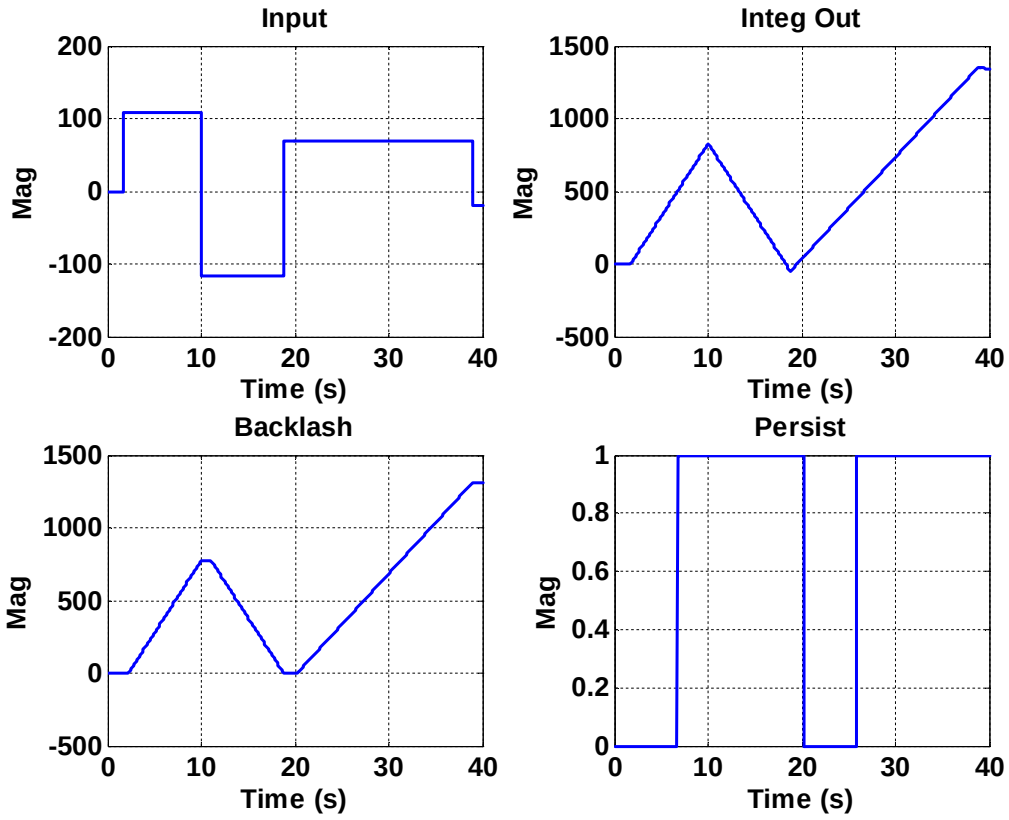

Figure 9. Input and Output of the persistence block

Matlab Code

The Benchmark Model was coded in Matlab. This was done to correlate the functional coverage metrics and the standard code coverage metric used in software testing today. The Matlab code was instrumented with tap out for coverage. All paths were covered by incrementing the specific COVERAGE array element. All conditions and decisions were covered making an array for the conditional coverage. If there is an AND logic with two inputs then a 4 element is used to monitor a FF, FT, TF and TT condition. An eight-element array is used for a 3 input logic. The combination of path and conditional coverage ensures that the complete code is executed.

Randomised Test Case Generation

A coverage metric has been defined for the block coverage. These metrics are dependent on the functional coverage of the block [1]. Random test cases were generated by injecting sinusoidal waveforms into the ten inputs. Each sine wave is defined by three parameters – frequency, amplitude and bias. These were randomly selected for each test case. The random waveforms were injected into the model and the coverage from the code and functional coverage were recorded for each run. After a set of 1000 runs the total coverage from all the runs were ascertained to be 100%. A selection process was carried out select the minim set of test cases, which provided 100% coverage. These are stored and used to find errors (mutants) in the model.

Mutation Runs

All the tests were executed on the mutant model compete.mdl. These mutants have been added manually for the training exercise. The trainees have to find the errors. They would be scored based on the probability of detection of the mutant as given in Table 2. The test cases fail to capture five mutants. Table 1 provides an analysis of the results

Table 1 Mutants not killed by the random tests

S No	Mutants	Reason
6	Backlash Bandwidth changed from 100 to 100.01	This is difficult mutant to catch. It is very essential that the input be very small to generate a waveform, which will step less than 0.01 to detect this small error.
8	Saturation Limit changed from 100 to 100.1	The saturation limit is correctly detected but to have its effect seen on

		the output requires small waveforms as the saturated output passes through an integrator in Model 5. It should pass through 4 more models before the effect can be detected.
10	Relational Operator changed from >= to >, comparing with 250.0	This is a difficult mutant to catch as the value should be exactly 250 to see the effect.
13	A third input added to the AND gate before Out4	This input is masked by the other inputs to the AND gate and its effect is not observable.
14	A ~= comparison added instead of > in persistence block	A ~= is the same as > in this instance and thus the mutant has the same effect as the original code making it undetectable.

A measure of the detection probability of the mutants was computed by simulating 11,000 runs. The percentage of test cases, which found the mutants, was the detection probability of the mutant. It is seen that the mutants not detected by the random test cases with 100% coverage have a detection probability less than 0.5% ranging to an absolute 0%.

Table 2 Detection Probability from 11,000 runs

S No	Mutation	DP %
1	NOT Gate on In8 changed to a NAND gate with inputs In6 and In8	85.3000
2	Constant8 changed from –1.0 to –1.001	91.8091
3	Added a spike in the Lookup Table as shown in Fig xx	1.1273
4	Extrapolate the Lookup Table instead of freezing at end values	7.9545
5	Rate limit changed from –10.0 to -10.0001	85.5909
6	Backlash Bandwidth changed from 100 to 100.01	0.1909
7	Discrete Time Integrator 2 algorithm changed from Forward Euler to Backward Euler.	15.0818
8	Saturation Limit changed from 100 to 100.1	0.4091
9	A gain of 0.1 added in the path after Discrete Filter 3	7.4455
10	Relational Operator changed from >= to >,	0

	comparing with 250.0	
11	A 1 frame delay added after In6	90.0364
12	A Filter initialisation changed with a different value for In2	91.9636
13	A third input added to the AND gate before Out4	0
14	A ~= comparison added instead of > in persistence block	0
15	The initial condition for the previous input changed in Rate limiter	90.9636
16	Initial value changed for Discrete Time Integrator 1	91.9636

Automated mutant generation

The model was used as a benchmark problem to verify random test case generation techniques. A novel method of test case generation using Taguchi was also used to generate test cases. An automated mutant generator was developed for the Simulink and Matlab code. These Matlab scripts generate all combinations of mutants for the Simulink and Matlab code. An OR gate, as an example, would be replaced by an AND gate, XOR gate etc in each mutant file. Each mutant file will have only one mutant. The mutant description is provided as a text file for the Simulink mutant models. The mutant Matlab code has the change description as the first line in the mutant. The script could generate 414 Simulink mutant and 7592 mutants for the Matlab code.

Results

The random test cases could kill 81.4% for the Matlab mutants and the Taguchi method could capture 88.64% of the mutants. The Taguchi method could kill 77.43% of the Simulink mutants and the random test cases 76.12% of the mutants.

List of files

SNo	File Name	Description
	compete.mdl	Model with the mutants
	compete_2010.mdl	Model original used for test case genertion
	compslp.m	Compute slope routine used by interpol1
	concover.m	Compute logical coverage
	CreateHarness.m	Create a test harness for the Simulink file
	Harness.mdl	The harness blank used by the CreateHarness.m file
	InsertMutants.m	Insert mutant into the Matlab code
	interpol1.m	Interpolation 1 D with coverage metrics
	kill_mutants.m	Script to run the stored tests on the Simulink mutants
	kill_mutants_m.m	Script to run the stored tests on the Matlab mutants
	model0001.m	Matlab code equivalent to the compete_2010.mdl file
	model0001_c.m	Matlab code as above but with coverage
	model0001_m.m	Matlab code as model0001.m but for mutant

		generation
	Original.mdl	The compete_2010.mdl model used by the randomiser mutant generator
	Randomizer.m	Script to generate mutants from Simulink models
	rand_input1.mat	Random test cases data. The 10 inputs are defined in these. These test provide 100% coverage.
	rand_input2.mat	
	rand_input3.mat	
	rand_input4.mat	
	rand_input5.mat	
	rand_input6.mat	
	rand_input7.mat	
	rand_input8.mat	
	results_rand1.mat	This is the model result from the random tests. This can be used for comparison instead of running the model again.
	results_rand2.mat	
	results_rand3.mat	
	results_rand4.mat	
	results_rand5.mat	
	results_rand6.mat	
	results_rand7.mat	
	results_rand8.mat	
	results_taguchi1.mat	Model results from test cases generated using a Taguchi Design of Experiments methodology.
	results_taguchi2.mat	
	results_taguchi3.mat	
	results_taguchi4.mat	
	results_taguchi5.mat	
	results_taguchi6.mat	
	taguchi_input1.mat	Test cases generated using a Taguchi Design of Experiments methodology. The 6 cases give the required coverage as the 8 random test cases
	taguchi_input2.mat	
	taguchi_input3.mat	
	taguchi_input4.mat	
	taguchi_input5.mat	
	taguchi_input6.mat	
	testcasegen.m	A Random test case generator. Can be used as example.

Reference:

1. Chethan CU, Yogananda Jeppu, Selvamurugan Hariram, Nagaraj Narayan Murthy, Prakash R Apte, “Input-Output Based Model Coverage Paradigm”, accepted IEEE Aerospace conference.