Mutation Testing for Model Based Requirements

By
Manjunatha L Rao, Jeppu Yogananda
Model-based design

- **Model-based design** is a development methodology for modern software artifacts.
- It promotes the use of powerful and specialized modeling languages, allowing the engineer to focus on the domain-specific aspects of the system under development.
- All flight control projects handled by Moog India Technology Center use Matlab/Simulink to model the flight control laws.
Model-based testing

- **Model-based testing** is the application of model based design for deriving and optimally executing the necessary test cases to perform software testing. Models are used to represent the desired behavior of the System Under Test (SUT).

- Moog India uses model based test techniques the independent verification and validation activities of flight control laws.

- We have qualified a set of Simulink blocks to be used for Model based testing
Mutation Testing

- **Mutation Testing** was originally proposed by Richard Lipton when he was a student in 1971. In a class term paper titled “Fault Diagnosis of Computer Programs” he proposed the initial concepts of mutation.

- [http://rjlipton.wordpress.com](http://rjlipton.wordpress.com)

Timothy Budd, RJ Lipton, R A DeMillo, F G Sayward, Mutation Analysis, Research Report #155, 1979

[http://rjlipton.wordpress.com](http://rjlipton.wordpress.com)
Mutation Testing

- **Mutation Testing** is a method of inserting faults into models based on requirements to test whether the tests pick them up, thereby validating or invalidating tests.
- The quality of the test suite is assessed by injecting mutations into the model and by measuring which percentage of these modifications can be detected when exercising the test cases.
- We say that a modification is detected if we can observe that the mutant and the original model generate different output signals.
- The resulting test vectors can be applied to validate the implementation of the design.
Indian LCA Testing

- Mutation testing has been used to generate test cases for the Indian LCA Control Law Testing
- This was done using a concept of delta model
- Delta model was the original Control Law Model with only one component perturbed. E.g. a filter coefficient perturbed in the 4\textsuperscript{th} decimal place
- The manually designed test case was considered as a good test case if it could pick up this error. More than 400 such test cases were generated to check each and every Control Law block

Goals

- To assess the quality of the tests by performing them on mutated model based requirements.

- To use these assessments to help construct optimized adequate tests

- To thereby produce a suite of valid tests which can be used on real programs
The Mutation Process

Requirements -> Test Inputs

- Mutant-1
- Mutant-2
- Mutant-N

Run Tests

- Any Mutants Left?
  - Yes: New Test Inputs
  - No: Test Complete

Any Mutations that are caught by tests are killed
A Requirement and It’s Mutants

- Original Requirement

- Mutants

   - Abs is replaced by a Gain Block
   - Abs is actually removed
An Example

<table>
<thead>
<tr>
<th>Input1</th>
<th>Input2</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-15</td>
<td>-15</td>
<td>0</td>
</tr>
<tr>
<td>-15</td>
<td>-20</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input1</th>
<th>Input2</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-15</td>
<td>-15</td>
<td>0</td>
</tr>
<tr>
<td>-15</td>
<td>-20</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>-30</td>
</tr>
</tbody>
</table>
Simulink blocks and its mutants

- The List of signs can be +-,-+,--
- The Operator can be OR, NAND, NOR, XOR
- The Relational Operator can be <, >, >=, ==, !=
- The Line connected to Booleans add NOT
- The Line connected floats add Gain or remove

And so on ...
Example Code

- load_system(modelname);
- **Load the Model**
  - if(strcmp(get_param(modelname,'Lock'),'on'))
  -    set_param(modelname,'Lock','off');
  - end
- **Unlock the model – This is important and required for libraries**
  - save_system(modelname, ['Mutant' num2str(count)],'BreakUserLinks', true);
- **Save it after breaking the links**
  - open_system(['Mutant' num2str(count)]);
  - rep = relation{i};
  - tmp = regexp(blockname,modelname,['Mutant',num2str(count)],'once');
  - set_param(tmp,'Operator',rep);
  - load_system(modelname);
- **Change the relational operator**
Example Code

- load_system(modelname);
- disp(['Mutant' num2str(count) ', Changed RelationalOperator in ' blockname ' Relation from ' get_param(blockname,'Operator') ' to ' rep]);

- Write out what has changed in a file so that you can do an analysis at the end of the run.
  
  - close_system(modelname);
  - save_system(['Mutant' num2str(count)]);
  - close_system(['Mutant' num2str(count)]);

- Close the system. Your Mutant model for Relational operator is ready
Video of Mutant Generation
Execution of Mutants

- To understand the capability of the test cases to catch mutants a batch of mutants are executed using the test cases
- The mutant model output is compared with the original model output for that test case
- If the error is above the threshold we say that we have killed the mutant
- If the error is below threshold we say that the mutant is alive
- If there is an execution error then we say the mutant is a dud! Use try catch to capture this
Killing Mutants
Results

- Many mutants are not caught!
- 11.85% of Logical mutants were not caught by the test cases. This is significant.
- Part of this percentage is due to the fact that the final output of mutated logical block would behave in the expected manner. Example replacing AND with NOTs and an OR logic.
- What is remaining is a definite cause of concern in a safety critical system. This percentage is around 2%.
- Can we live with this?
Results

- In the Ratelimiter block \(<\) LowerLimit was replaced by \(\sim=\) LowerLimit. This produced a very small error of 0.02 when compared with the original.
  - What should be my pass fail criteria?
  - What can I do to my input profile (if anything possible) to increase this threshold?

- Floating point comparison “with equal to” is very difficult to catch. This mutant is never killed.

- Some blocks mask errors in the previous blocks. In an integrator block output a major error cause only a 0.0040 error in the later block outputs.
Best Practices

- The 2% of mutations not being killed is a serious situation. Testing alone is not sufficient to prove that the code is safe for use.
- Code reviews, coverage metrics are a **MUST** to supplement test results.
- “Program testing can be used to show the presence of bugs, but never to show their absence!”

Edsger Wybe Dijkstra (1930-2002)
Best Practices

- Automation of testing is a must for the time critical situation today. Define your pass fail criteria with sound mathematical/experimental foundation before you press the button to start testing. Remember some major errors have a very small effect on the final output.

- If possible tap out more variables during testing. Add additional “tapouts” and compare these with the model outputs. Simulink has an excellent capability to tapout any signal. Use a simple script to automate this process. Use a similar practice in the code!
Conclusions

- Mutations injected into models provide an ideal way of verifying the efficacy of the testing.
- Matlab and Simulink provide enough commands and capabilities to easily automate this process.
- We require several benchmark problems on the Mathworks website which can be used for empirical and experimental software research.
Thank You