PROGRAMMED TEST CASE GENERATION FROM SIMULINK/STATEFLOW MODEL

Ravikant Sharma

Computer Science & Engineering Department, National Institute of Technology, Rourkela
Rourkela, Odisha769008, India
213CS3177@nitrkl.ac.in

Surya PrakashSonwani

Computer Science & Engineering Department, Rajiv Gandhi Technical University, Bhopal
Bhopal, Madhya Pradesh462036, India
suryasonwani640@gmail.com

Abstract

Matlab, Simulink/Stateflow Model is the most extensively used industrial tools that include system development that allows models to be developed, visualized and exercised. Matlab, Simulink/Stateflow (SL/SF) is used particularly for developing embedded system. Usually SL/SF models are hierarchical;we can split up any complex system into trivial parts that can be viewed as a level of abstraction that helpful in understanding the system easily. The resulting model must be tested in order to find fault in the system, especially embedded system. In order to doing the model based testing of the SL / SF model, we have primarily developed any SL/SF system, subsequent we have to convert that model into xml file, succeeding using XML parser and our proposed algorithm we have to develop SL/SF System dependency graph (SSDG) for SL/SF model, subsequent using SSDG, we apply our algorithm to generate test sequences and test cases for the SL/SF model.

Keywords*:* SL/SF model; SSDG; Model based testing; Test Sequences; Test cases.

1. Introduction

Testing is a significant phase in a software development process to build the quality product. Manual testing consumes a lot of time and cost. Testing process consists of the three tasks: Test case generation, execution, evaluation.

Among the three, test cases generation problem is the most difficult task. A test case is a triplet (I, S, O) where I is data input, S is state, and O is the expected output, however automatically generating test cases from a code is a very thorny task. So most of the researcher tries to generate the test cases from the model of the system requirement.

Embedded Control Systems are now integral parts of many application systems in the areas, of Aerospace, Communication, Automobiles, etc. As a result, scientists and engineers are looking for easy and trustworthy techniques to design, develop, test and verify these systems. With a model based design and development becoming a drift, industries use design and simulation tool sets like MATLAB and Mathematica.

MATLAB, Simulink/Stateflow (SL/SF) [1] is a high level model designing tool very popular in many industrial application domains. It enables modeling, simulating and analyzing dynamic systems. It provides a wide range of library blocks, for example, Math Operation blocks, Logic and Bit Operation blocks, Signal Routing blocks, to name a few. Systems can also be multi rate, i.e., have different subsystems that are sampled or updated at different rates. Simulink having the blocks of libraries which contain integration, summation blocks.

To capture the discrete control states, one generally uses Stateflow which is a component of Simulink. Stateflow is an interactive graphical design tool. It provides a graphical editor on which the Stateflow graphical objects can be dragged and dropped from the design palette can be put to create finite state machines. It allows hierarchical state machine diagrams, State charts to be combined with flow charts. SL/SF is a widely accepted tool in the industry for model based development of systems. This environment supports the hierarchical development of complex design controller and also provides a rich set of high level and customizable computational and control blocks suitable for hybrid control systems. A wide variety of application specific block-sets available with SL/SF environment enable easy development of control systems in various domains. SL/SF models help in the early design exploration, simulation, automatic code generation for different hardware/software platforms.

Figure 1 shows an example ofStateflow model, which represents the power switch. A sample of the Simulink model which containing a Stateflow diagram in it. When you simulate this model, the generation of the input event from Simulink, Switch, will toggle the activity of the states between Power_on and Power_off. In this model Simulink used as an interface for the Stateflow model.

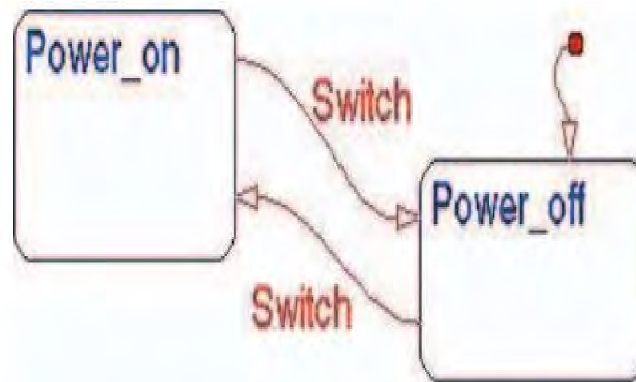

Fig.1. An example of State flow model.

2. Proposed Work:

Our proposed work is based on the graph called Simulink/Stateflow dependency graph (SSDG), in which nodes represent the blocks of the SL/SF model and edges represents the dependencies between blocks. The overall algorithm of our work is as follows:

2.1 Overall Algorithm of our approach

Step1: Draw Simulink Model by using MATLAB, Simulink tool and Stateflow model is added to the Simulink model by using the MATLAB Stateflow design tool. (It creates.mdl file)

Step 2: Generate XML file from Simulink Model(.mdl)generated above.

Step 3: Read the Blocks of model (in Java using.mdl file path as input) and using an xml parser (xml file as an input), generate an adjacency matrix that contains a dependency amongst the blocks of SL/SF model and store on dotty file.

Step 4: Using dotty file generated in Step3, generate an intermediate graph called Simulink / Stateflow Dependency graph (SSDG) using GraphViz tool.

Step 5: Generate test Sequences using the intermediate graph by applying the approach Depth first search (DFS) in the graph.

Step 6:Next, for each test sequences generate a test case using intermediate graph and dotty file.

Figure 2 shows our proposed methodology and step by step procedure for generating test cases and test sequences using dependency graph. Firstly, we have to create SL / SF models using Matlab, Simulink library tool, save it to .mdl file. Next we have to convert that .mdl file into xml file. Nowusing our proposed algorithm dependency_Graph_Generation that takes .mdl file and xml file as input and generates SSDG using the GraphViz tool (that takes a dotty file as an input). Next, apply different testing coverage criteria like state coverage and transition coverage to generate test sequences by applying depth first search approaches (DFS). Next, for each test sequences generate set of test cases.

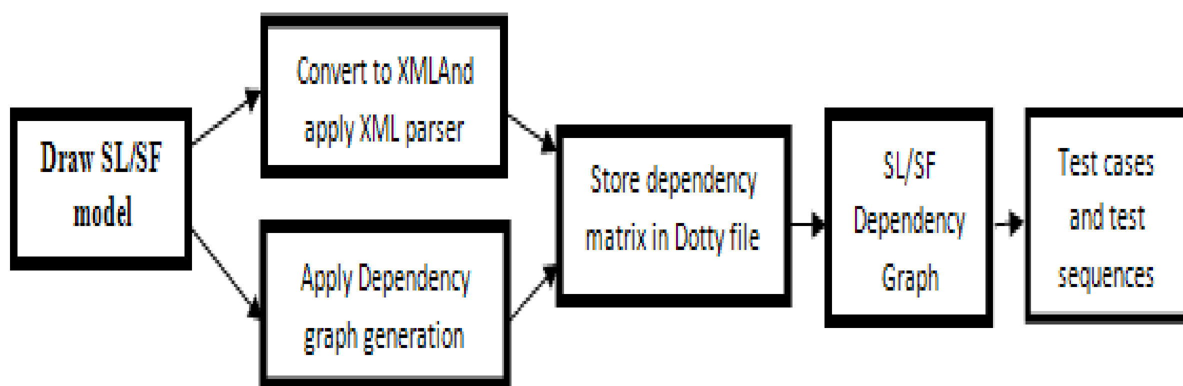

Fig. 2. Proposed Methodology

2.2 Algorithm: Dependency_Graph_Generation

***Input*:**Xml file and mdl file path of SL/SF model.

Output:Dependency graph

Startmain

//First, create an SL / SF models using Matlab SL/SF design Tools and save it. (It creates .mdl file)

Take .mdl file path and xml file as an input and generate model object and passed it to the function graphGeneration.

For all up to all block present in the SL / SF model do

- Read each block.
- Obtainall theneighbor next block of present block.
- Write the next block in the matrix form and store that into dot file.
- If any present block contains Simulink/Stateflow subsystem then

Push that block into the queue.

End if

End for

For all up to all block present in the queue, i.e. queue is not empty do

- Read each block.
- Obtainall theneighbor next block of present block.
- Write the next block in the matrix form and store that into dot file.

End for

Now using GraphViz Tool, take above dot file generated in the above step as in input to the tool.

Generate Graph using this tool. This Graph is Called SL/SF dependency Graph.

End main

3. Implementation and Result:

In this section we have taken an example of case study electric fan for generating test cases, test sequences from a Simulink / Stateflow model for electric fan. So let's discuss our implementation of case study step by step:

3.1 Construction of Simulink/Stateflow model: UsingMatlab Tool and Simulink library, we first have to develop Simulink/Stateflow model by drag and drop from the design panel of Simulink library. Since in the Simulink library Stateflow design panel is present, which further contains blocks, states, transition, etc. are present for developing Stateflow model and running any one either Stateflow model or Simulink model, automatically both runs simultaneously that is reason why generated model is called a Simulink/Stateflow model. Its file extension is .mdl (model description language).

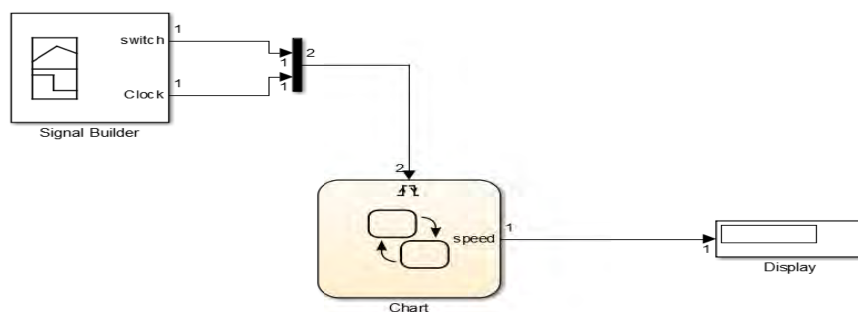

Fig.3.Simulink/Stateflow Model for Electric fan Example.

Fig. 3 shows the Simulink/Stateflow model for the case study electric fan. Signal Builder is used in model for generating the signal for electricfan case study,we have used Chart that represents the state chart for fan model which further be subsystem into states and transitions.One scope block is used at the end for displaying the result.

Fig. 4 represents the state chart for the electric fan, which implements the chart in a Simulink model. It represents the different states of the system goes during the course of its execution. It contains mainly two states named “on” state and “off” state. Since every state chart model contains the default transition that shows from where execution starts. Here default transition is “off” state. “On” state is further divided into four sub states

named as “one”, ”two”, ”three” and “four ” states. It contains one variable name speed, which is used in manipulatingbetween different states.

Off state:It is a default state which contains the entry section in which speed is set to 0. Anytime, ifstate “one” found the condition [off] to be true, then control goes to “Off” state.

On state: When “Off” state finds the condition [speed==1] then transition control enters into the “On” state which further contains four sub states in which state “one” is default transition.

One state: Once “On” state is active, control goes to state “one” because the default transition of “On” state is state “one” and also when state“two” found the condition [speed==1] to be true, then control goes to the state “one”. It contains entry section in which speed is set to 1.

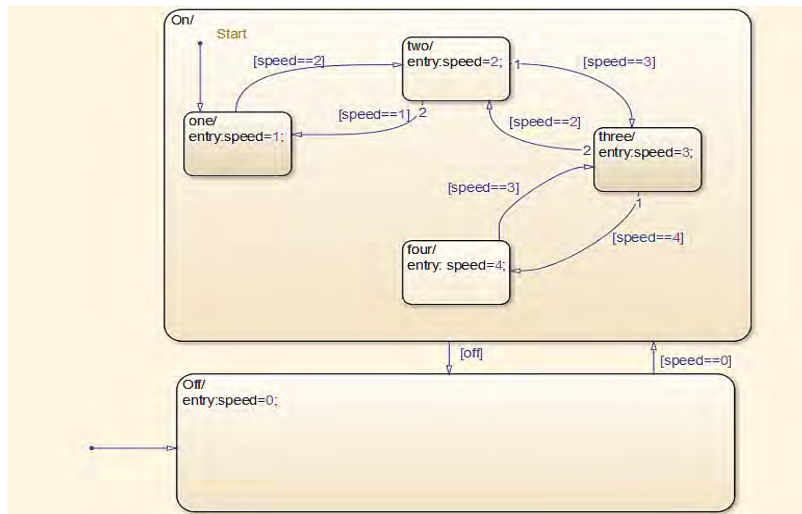

Fig. 4. State chart model for Electric Fan

*Two state:*Once the state “one” found the condition [speed==2] are to be satisfied, then controlgoes to state“two” and also when state “three” found the condition [speed==2] to be true, then control goes to state“two”. It contains entry section in which speed is set to 2.

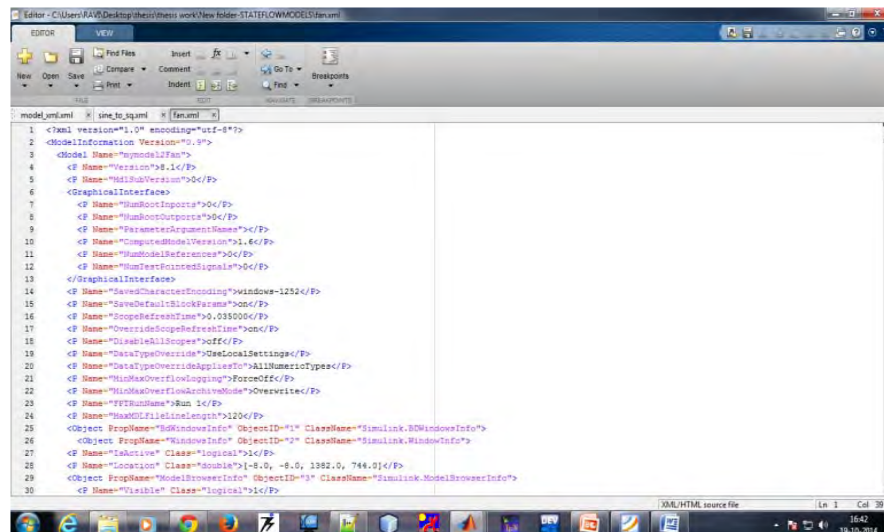

Fig. 5. Xml file of Electric Fan

Three state:Once the state “two” found the condition [speed==3] is to be satisfied, then controls goes to state“three” and also when state“four” found the condition [speed==3] to be true, then control goes to state “three”. It contains entry section in which speed is set to 3.

Four state: Once the state “three” found the condition [speed==4] is to be satisfied, then controls goes to state “four”.It contains entry section in which speed is set to 4.

3.2 Convert to XML: After developing the SL/SF model in Matlab. It generates .mdl file. Next step is to convert that .mdl file into XML file using a command in Matlab.Fig. 5. Shows the converted xml file of electric fan case study.

3.3 Generation of Dependency Graph:Nextstep is to apply our proposed algorithm of dependency graph generation that takes xml file and .mdl file as an input and gives intermediate graph generation as an output.

This algorithm takes .mdl file path as an input. After creating model objects for the SL / SF model and passes.mdl file path to the function graph_Generation and apply xml parser which takes an xml file as an input to xml parser. Next we have to apply loop up to all block covers in the model and within this loop, we have to perform these operations: read each block and extract the information of each block, then obtain all the next neighbor block of the current block, then Write the next block in the matrix form and store that into dot file. If any present block contains SL/SF subsystem, then push that block into the queue. After completing of first loop, we have to check whether the queue is empty or not, if queue is not empty, then again apply one loop up to block available in the queue or queue becomes empty and perform the following actions: read each block and extract the information of each block, then obtain all the next neighbor block of present block, then write the next block in the matrix form and store that into dot file. Fig. 6 represents the dotty file that store blocks dependency which is further used to generate the dependency graph. GraphViz tool is used to generate dependency graph using generated dotty file. Fig. 7 represents the SL/SF dependency graph for model electric fan.

```
strict digraph graphname {
"start/"->"off/"[label="[speed==0]"];
"off/"->"on/"[label="[speed==0]"];
"on/"->"one/"[label="[speed==1]"];
"one/"->"two/"[label="[speed==2]"];
"two/"->"three/"[label="[speed==3]"];
"three/"->"four/"[label="[speed==4]"];
"four/"->"three/"[label="[speed==3]"];
"three/"->"two/"[label="[speed==2]"];
"two/"->"one/"[label="[speed==1]"];
"one/"->"on/"[label="[speed==0]"];
"one/"->"off/"[label="[off]"];
"two/"->"off/"[label="[off]"];
"three/"->"off/"[label="[off]"];
"four/"->"off/"[label="[off]"];

}
```

Fig. 6. Dotty file generated for Electric fan model

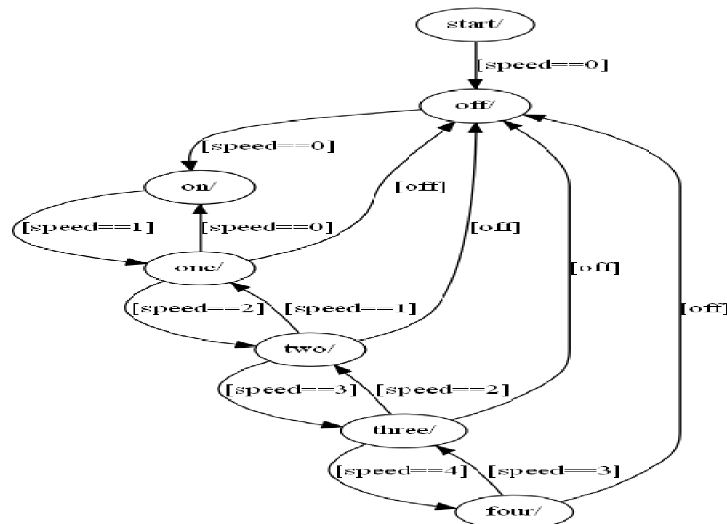

Fig. 7. SL/SF Dependency Graph for Electric Fan

3.4 Test Sequence and test case Generation:Next step is to generate the test sequence from the generated dependency graph by considering state coverage approach.

*State Coverage:*The coverage, which covers states in all the possible ways in the dependency graph is called state coverage.

For generating the test sequences, we have performed the DFS approach to generate the test sequences. The test Sequences for following case study of electric fan are:

1) Start→off

2) Start→ off→on→one→off
3) Start→ off→on→one→two→off
4) Start→ off→on→one→two→three→off
5) Start→ off→on→one→two→three→four→off
6) Start→ off→on→one→two→three→two→off
7) Start→ off→on→one→two→one→off
8) Start→ off→on→one→two→three→four→three→off

Next, for each test sequence we have to generate the test cases using the test sequences and dependency graph. Table 1 represents the set of test cases for SL/SF model.

Table 1. Test Cases for Electric fan model

Test ID	**Current State**	**Test Cases**	
		Input Condition	**Expected Output**
1	Start	[Speed == 0]	Off
2	Off	[Speed == 0]	On
3	On	[Speed == 1]	one
4	One	[Speed == 2]	two
5	Two	[Speed == 3]	three
6	Three	[Speed == 4]	four
7	Four	[Speed == 3]	three
8	Three	[Speed == 2]	two
9	Two	[Speed == 1]	one
10	One	[Speed == 0]	on
11	Four	[off]	Off
12	Three	[off]	off
13	Two	[off]	off
14	One	[off]	off

4. Comparison with related work:

Many different approaches are there related to our work.One approach is T-Vec[2] tester that delivers a comprehensive approach for test generation for Simulink model that offers an exhaustive solution for continuous model analysis, test execution and automatic test generation. It drills the path boundaries for generating test vector for path throughout the model hierarchy. Unreachable paths that result in dead code are identified and hyperlinked to the Simulink model elements involved. It is based on the assumption that if there is no coincidental correctness, then test cases that limit the boundaries of domains with arbitrarily high exactitude are adequate to test all the points in the domain. But one disadvantage is that it does not consider the Stateflow of the Simulink.

Zhan and Clark[3] proposed a technique called tracing and deducing, that enhanced the capability of search-based test data generation for Simulink.

Disadvantage of Zhan and Clark approach: They don't consider a state problem. The state problem remains a challenge for higher level as well as code level test data generation. Automatic generation of test data for higher level models more generally is a very challenging (performance, reduces as complexity of model increases).

Nayak [4] proposed a methodology Meta model for Simulink/Stateflow called as the Simulink dependency Graph (SDG). The SDG captures all implicit dependencies between different blocks of the SL/SF model and that represents them explicitly, thereby making it possible to perform several types of analysis on the SL / SF model.

MirkoConard [5] et al proposed an approach that designs a test suite for code generation tools. They describe the design of a test suite for code generation tools. This method provides solutions of different problems of different types of tools that gives how the correct transformation of a source language/model into a target language can be proved. The application of the proposed testing approach leads to a generation of sets of test suite, which is suitable for testing code generators systematically.

But, the existing code generators can’t guarantee that the automatically generated code from tool, compiles correctly as mentioned in the design due to the following reasons:

1. Errors in the Simulink/Stateflow diagram nodes will get carried over.

2. Errors in the automatic code generator for the Simulink/Stateflow diagram caused for example by finite precision arithmetic or timing constraints.

3. Any human errors in the selection of code generation options, library naming or inclusion, and others.

But our approach overcomes these limitations, no need to generate code from the models in our approach because of that it overcome the MrkoConard's approach. We also cover all the blocks and all transitions through the generated graph so that our proposed approach over came from these limitations. The Zhan's approach also not covering all the Blocks due to small signal generation, but our approach overcomes this limitation also.

5. Conclusion

We proposed a methodology to generate test cases from SL/SF models. First, weconstruct the model in the MATLAB environment by using Simulink/Stateflow designing tool. By simulation, we verify the model. After verification by using our approach we generated a graph. From that graph we performed the traversal operations and generated the test sequences. The test sequences are used to generate test cases. This approach covers much important coverage like state coverage, transition coverage. This is more accurate than the methods which are generating test cases using code generation.

References

[1] The MathWorks, Mathworks MATLAB Simulink. "http://www.mathworks.com/products/simulink.

[2] T-vec," Website. http://www.t-vec.com/

[3] Y. Zhan and Clark, A search-based framework for automatic test-set generation for Matlabsimulink models," Software Eng. SE-10, vol. PhD thesis, December 2005.University of York.

[4] SurajNayak. "A Metamodel for Simulink/Stateflow models and its applications", M. Tech. Thesis, IIT Kharagpur, Computer Science Department (2013).

[5] I. Sturmer and M. Conrad, "Test suite design for code generation tools," in Automated Software Engineering, 2003. Proceedings.18th IEEE International Conference on, pp. 286-290, IEEE, 2003.

[6] N. Vamshi Vijay, "Regression test selection based on analysis of Simulink/Stateflow models", M.Tech. Thesis, IIT Kharagpur, Computer Science Department (2012).

[7] A. Windisch, "A Search based testing of Simulink models containing stateflow diagrams,"IEEE Trans., vol. Software Engineering Companion Volume, pp. 395-398, 2009. Daimler Center for Automotive IT Innovations DCAITI, Tech. Univ. Berlin, Berlin, Germany.

[8] R. Systems, "Reactis simulator / tester." Website. http:www.reactive-systems.com.

[9] A. A. Gadkari, S. Mohalik, K. Shashidhar, A. Yeolekar, J. Suresh, and S. Ramesh, "Automatic generation of test-cases using model checking for SL/SF models," in 4thMoDeVVa workshop Model Driven Engineering, Verification and Validation, p.33, 2007.

[10] R. Alur, A. Kanade, S. Ramesh and K.C. Shashidhar, “Symbolic analysis for improving simulation coverage of Simulink/Stateflowmodels”.Proceedings of the 8th ACM international conference on Embedded software P. 89-98, 2008.