Model-Based Test-Case Generation for Simulink/Stateflow using Dependency Graph Approach

[1]Adepu Sridhar, [2]Srinivasulu D and [3]Durga Prasad Mohapatra
Department of Computer Science and Engineering
National Institute of Technology
Rourkela, India
[1]sridharuce@gmail.com, [2]cnud916@gmail.com and [3]durga@nitrkl.ac.in

Abstract—**Testing is an ultimate phase of product life cycle to which particular attention is paid, namely when dependability is of great importance. Modeling technology has been introduced into the software testing field. However how to carry through the testing modeling effectively is still a difficulty. Based on the combination of simulation modeling technology and dependability we have proposed an approach to generate test cases. In our approach, first, the system is modeled in MATLAB using Simulink/ Stateflow tool. After the model creation we verify that system and generate a dependency graph of that system. From that graph we generate test sequences.**

Keywords-Simulink/Stateflow; dependency graph; test sequences ; software testing

I. INTRODUCTION

Embedded Control Systems are now integral parts of many application systems in the areas of Aerospace, Communication, Automobiles, Commercial (computer peripherals, appliances, etc.), Industrial (machinery) etc. As a result, everyone looking for easy and reliable techniques to design, develop, test and verify these systems. With a model based design and development becoming a trend, in industries as well as academia. To implement these systems and analyze models everyone normally use Mathworks Simulink tool [1]. Simulink models consist of functional blocks with input and output ports to interconnect them. These interconnections define the data flows between the blocks. Embedded Systems are usually state based, they can be situated in various states depends on the inputs and the control logic.

The model must be tested in order to detect faults in the embedded system as early as possible. Exhaustive test is not possible for any system. The test data generation process is very costly and time consuming and error prone when done manually, the automation of this process is highly required [3]. Embedded controller software is usually based on various states and for that states representation often uses Stateflow [2] diagrams utilization of the internal structure of the diagram to generate Test cases is important. This is achieved by covering Stateflow coverage and particular state coverage.

Code generators are used within the Simulink/Stateflow (SL/SF) to automatically generate the embedded software for the target system from the Simulink/Stateflow diagram. The existing code generators cannot guarantee that the generated code complies correctly as mentioned in the design. Verification and testing is necessary to find errors in the code generation process those are helpful to avoid software faults. Several types of errors may occur in the implementation process from the Simulink/Stateflow diagram of the target code, such as:

- Errors in the Simulink/Stateflow diagram nodes will get carried over.
- Errors in the automatic code generator for the Simulink/Stateflow diagram caused for example by finite precision arithmetic or timing constraints.
- Any human errors in the selection of code generation options, library naming or inclusion, and others.

II. BASIC CONCEPTS

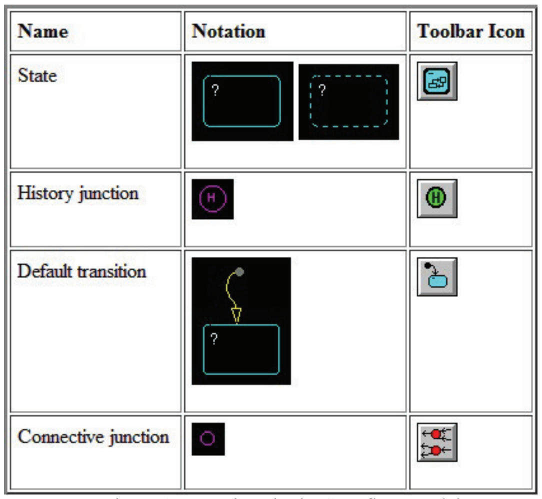
Simulink is an environment for multidomain simulation and Model-Based Design for dynamic and embedded systems. It is an interactive graphical environment and a customizable set of block libraries that let us design, simulate, implement, and test a variety of time-varying systems, including communications, controls, signal processing, video processing, and image processing. Simulink is used for modeling both continuous and discrete Systems. Simulink blocks are divided into different types according to their behavior.

- The Source library: Contains blocks that generate signals.
- The Sink library: Contains blocks that display or write block output.
- The Linear library: Contains blocks that describe linear functions.
- The Nonlinear library: Contains blocks that describe Non-linear functions.
- The Connections library: Contains blocks that allow multiplexing and DE multiplexing, implement external

Input/Output, pass data to other parts of the model, create subsystems, and perform other functions.

A Stateflow diagram is a graphical representation of a finite state event driven machine. Stateflow is a powerful graphical design and development tool for complex control and supervisory logic problems.

- Visually model and simulate complex reactive systems based on finite state machine theory.
- Design and develop deterministic, supervisory control systems.
- Easily modify your design, evaluate the results, and verify the system's behavior at any stage of your design.
- Automatically generate integer or floating-point code directly from your design.
- Take advantage of the integration with the MATLAB and Simulink environments to model, simulate, and analyze your system.



Name	Notation	Toolbar Icon
State	? ?	
History junction	H	H
Default transition	?	
Connective junction		

Figure 1. Notations in the Stateflow Model

Stateflow allows us to use flow diagram notation and state transition notation seamlessly in the same Stateflow diagram. Flow diagram notation is essentially logic represented without the use of states. Stateflow uses a variant of the finite state machine notation established by Harel [1]. Stateflow allows two basic building blocks states and transitions to represent the finite state machine. Stateflow enables the representation of hierarchy, parallelism, and history. Hierarchy enables you to organize complex systems by defining a parent object structure. For example, you can organize states within other higher-level states. A system with parallelism can have two or more orthogonal states active at the same time. History provides the means to specify the destination state of a transition based on historical information. These characteristics enhance the usefulness of this approach and go beyond what STDs and bubble diagrams provide. The Stateflow machine is the collection of Stateflow blocks in a Simulink model. The Simulink model and Stateflow machine work seamlessly together. Running a simulation automatically executes both the Simulink and Stateflow portions of the model. Simulink model consists of Simulink blocks, Subsystems, toolbox and Stateflow Block. In the Simulink model Stateflow represented as Chart. A collection of all these charts is called as a Stateflow Machine. Chart consists of state, transition, data, events are there. The graphical objects in the Stateflow model are State, History junction, Default transition, Connectivity junction. These notations are taken from MathWorks [2] standard library. These are shown in Figure 1.

Stateflow diagram example: Figure 2 shows the small Stateflow model example 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. an In this model Simulink used as interface for the Stateflow model.

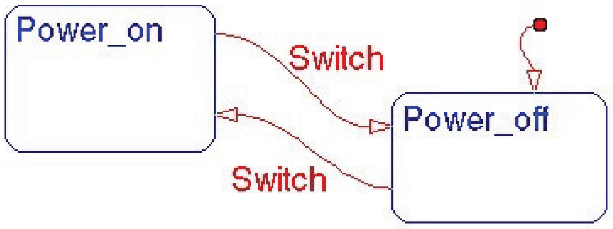


Figure 2. An example of Stateflow model

In the Stateflow logical operations are performed with four actions:

- Entry actions
- During actions
- On Event actions
- Exit actions

III. PROPOSED APPROACH

Our method of generating test cases is based on using graph. The edges in the graph have special meaning based on the pair of nodes that connect. We represent an edge in the graph using a node pair. In Simulink model the edges represent the interconnections between the Simulink blocks. An interconnection between two Simulink blocks represent the transfer of a signal from one block to the other, one dependency exist that is output dependency.

Output dependency edge: Output dependency graph is a graph which contains the edges like, edge from A to B means that block B is output dependent on block A, input for the block B depends on the output produced by block A. In Stateflow the dependency exists is control dependency.

Control Dependency: A control dependency graph is a graph, which represents an inter stage dependency that arises due to a change of state in the Stateflow model.

Graph construction is based on the model information present in the mdl file of a given model. First we discuss the overview of our approach. This is followed by an algorithmic representation of the same. First the SL/SF model is used to construct a top level Graph. In the top level graph, every subsystem is represented using a single node. Every subsystem and with respect to that subsystem, Stateflow are represented using separate Graphs. The number of graphs are constructed based on the hierarchy of subsystems and Stateflow in a SL/SF model.

Algorithm Details: This graph takes the SL / SF model as input and generate graphs. First, the top level graph is constructed. If any subsystem is found to exist in the model then the graphs for the subsystems are constructed. After which the “Stateflow Graph Construction" is called to generate graphs for Stateflow charts. Whenever we are constructing the graph we make a matrix for traversing purpose.

1) Algorithm1:Dependency_Graph_Construction

```
Input: file path // path of mdl file of a model
Output: Output dependency graph of the Simulink model
Begin
//create a model object
Read the blocks from the model object
// read the each block
while up to all blocks cover do
extract all adjacent blocks of current block
Note down the adjacency in the adjacency matrix
write the adjacency matrix information to the dotty files
if any block having Simulink subsystem then
push that block to the Queue
end if
end while
while Queue is not empty do
Read the each block
extract all adjacent blocks of current block
Note down the adjacency in the adjacency matrix
write the adjacency matrix information to the dotty  files
end while
Stateow graph construction(file path);
End
```

2) Dependency_Graph_Construction:

The Dependency Graph Construction algorithm is taking the file path as input. First, we are creating a model object. Second, read the blocks from the model object. For every block considers the adjacent blocks and note down in the adjacency matrix. Using the adjacency matrix draws the graph in the Dotty file using Graphviz [12] tool. When traversing the block if any block is a subsystem push it to the Queue. After all the completion of block traversal, check the Queue and if it is not empty, construct graph for the every subsystem. At the end call the Stateflow Graph Construction.

3) Algorithm2: Stateflow Graph Construction

```
Input: file path
Output: control dependency graph of the Stateflow model
Begin
Get the Stateflow machine
Get the charts from the Stateflow machine
Read every Stateflow chart
while up to cover all charts do
Read all the states present in a model
while states not empty do
Read each nodes from the state
Note that adjacency in the adjacency matrix
write down the adjacency matrix to the dotty files
end while
end while
End
```

4) Stateflow_Graph_Construction:

The Stateflow Graph Construction algorithm is taking the file path as input. First, we get the Stateflow machine, From the Stateflow machine we get the Stateflow charts. Every chart is a subsystem. Read each node from the every chart. Note the adjacency in the adjacency matrix. Using the adjacency matrix draws the graph in the dotty file using Graphviz [12] tool.

IV. IMPLEMENTATION

In this section, we explain the working of our algorithm by taking the Fan example using a Simulink/Stateflow model, as described below and it is shown in Figure 3.

In this example the fan receives signal from signal builder. According to that signal the fan rotates. While rotating it changes from one state to the other state. First the control starts in the default state. In our example Off/ is the default state. At this time the speed of the fan is `0'. When the switch signal becomes ‘on' the transition goes to work and control goes to On/ state. On/ state having hierarchy, it is having sub states one/, two/, three/, four/. In these states the default state is one/, this time the speed of the fan is ‘1’. When the clock signal is on one/ state outline transition goes to work and control goes to two/ state. Every time whenever clock changes from off to on speed changes with respect to the states. Whenever the switch signal goes to off then control goes to Off/ state. This is shown in Figure 3. When drawing the graph we are representing the default transition by using the START node.

1) Implementation Details:

1. Language used for implementation is Java.

2. Input: SL/SF MATLAB model file having .mdl extension.

3. Output: Graph representation of the model and test sequences.

2) Tools used for the implementation:

1. Netbeans

2. GRAPhviz.

3. Matlab Simulink.

The Figure 3 is the Stateflow in the SL/SF model. When we are passing this model as input our algorithm generates the graph which is shown in Figure 4. In the On/ node it is having sub nodes for the representation of sub nodes. Our algorithm generates one more graph which is shown in Figure 5. From this graph we will generate test sequences based on the state coverage, transition coverage etc.

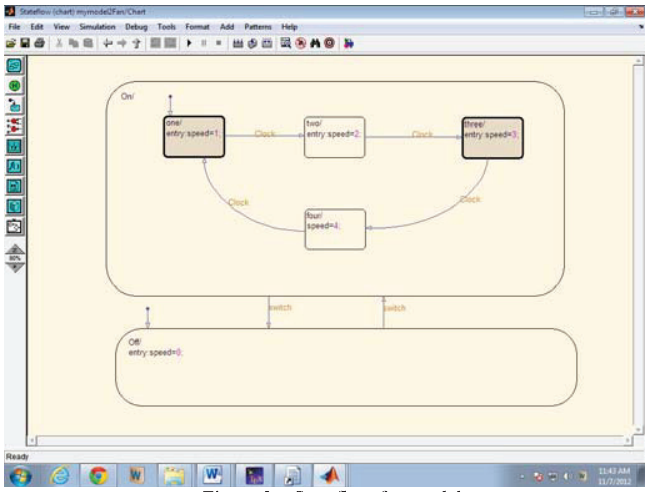


Figure 3 . Stateflow fan model

3) State coverage: The coverage which covers states in all the possible ways is State coverage.

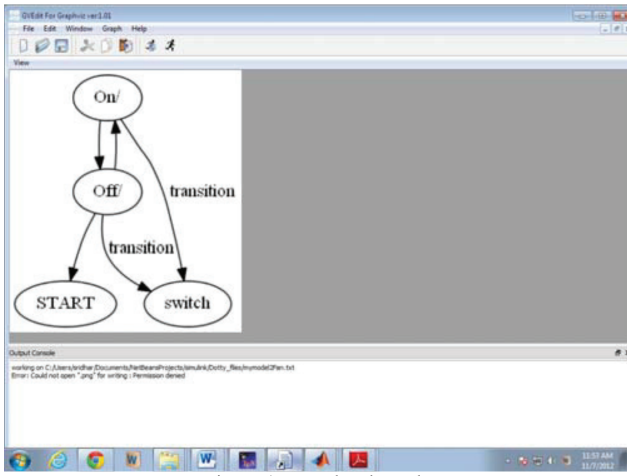
4) Top level test sequences:

1. Start -> off/ (if the signal is not generated)

2. Start-> off/ -> on/-> off/(if the signal is generated)

5) Secondary level test sequences: It is within the On/ state so it will execute whenever the signal generated successfully.

1. start-> one/-> two/-> three-> four/-> one/ (In this case it is covering all states)



```
working on C:/Users/sridhar/Documents/NetBeansProjects/simulink/Dotty_files/mymodel2Fan.txt
Error: Could not open ".png" for writing : Permission denied
```
Figure 4 . Top level graph

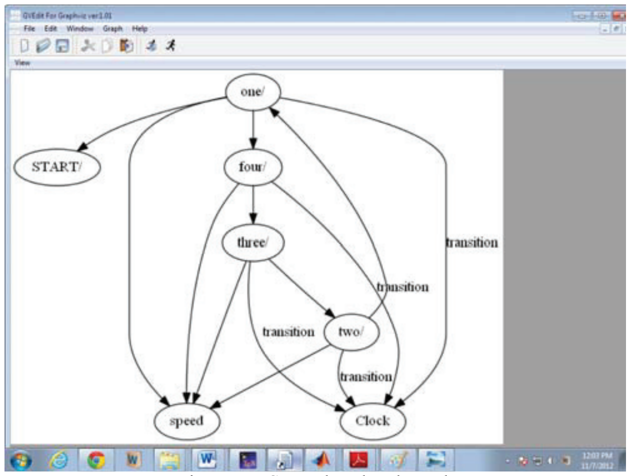


Figure 5 . Secondary level graph

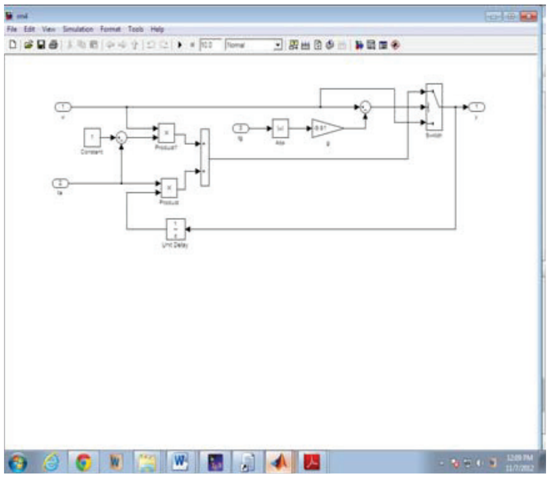


Figure 6. Example of Simulink model

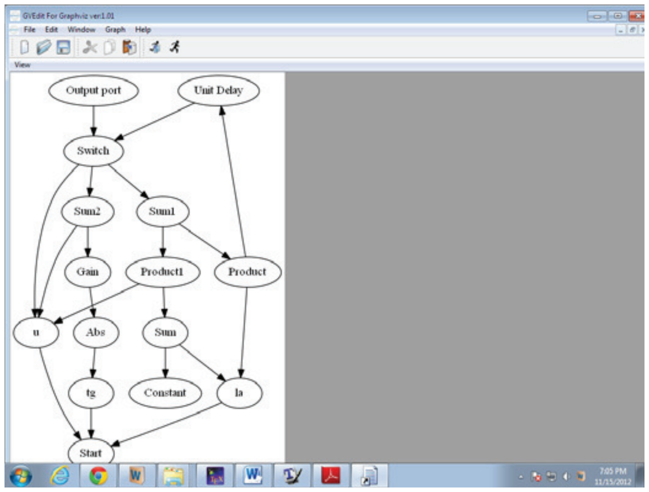


Figure 7 . Graph of Simulink model

6) Transition coverage: The coverage which covers transitions in all the possible ways is Transition coverage.

- **Top level test sequences**:

1. Start -> off/(if the signal is not generated)

2. Start-> off/-> switch-> on/-> switch-> off/ (if the signal is generated)

- **Secondary level test sequences:**

1. Start -> one/ -> clock->two/->clock->three/->clock-> four/ -> clock (in this case it covers all states and all transitions)

The transition coverage is better than the state coverage it covers all the states and extra it covers all transitions. We generated test sequences from Simulink model that is shown in Figure 6. When we are sending this model as input to the our algorithm the the graph is coming out. That is shown in Figure 7.

From Figure 7 the test sequences are:

1. Start-> tg -> Abs -> Gain ->sum2 ->switch -> output port->Start -> u -> switch -> output port

2. Start -> u->sum2 ->switch->output port->Start -> u -> product1 -> sum1-> switch->output port

3. Start -> la -> sum -> product1 -> sum1 -> switch -> output port->Start -> u ->switch -> unit Delay -> product -> sum1 ->switch -> output port

4. Start -> u -> sum2 -> switch ->unit delay -> product -> sum1 ->switch ->output port->Start -> tg -> Abs -> Gain -> sum2 -> switch ->unit delay -> product -> sum1 -> switch -> output port

5. Start -> u -> product1 -> sum1 -> switch -> unit delay -> product -> sum1 ->switch -> output port -> Start -> la -> sum -> product1 -> sum1 ->switch ->unit delay -> product -> sum1 -> switch -> output port

6. Start -> la -> product -> sum1 -> switch -> unit delay -> product -> sum1 ->switch -> output port

V. COMPLEXITY ANALYSIS

The complexity of the model under test has a big influence on the success of the test data generation process due to associated increasing execution times. We now analyze the time complexity and space complexity of the graph construction procedure. First we consider the complexity for construction of graphs of the models having only Simulink blocks with no Stateflow charts. Next we analyze the complexity of construction of graph for Stateflow charts alone and then generalize for the models having both Simulink and Stateflow blocks. First assume a simple model having no hierarchy, that is only one level and has 'n' blocks on that level. Each block in the model is visited once to extract its information into adjacency matrix, hence O(n). This adjacency matrix is used for graph traversal and construction. Adjacency matrix has nXn entries, accessing adjacency matrix has time complexity O(n*n). So the time complexity of the above such models will be O(n2). Let us take complex model having `m' subsystems in the top level and also each level there exist 'n' blocks, this would result in (1+m(k+1)) graphs. One block for the top level having B blocks out of which `m' are subsystems and each subsystem in turn has `k' subsystems, there would be `k' graphs plus one more graph having these 'k' subsystems. Total subsystems are m(k+1) graphs. Finally, including the top level total graphs would be 1+m(k+1). Total complexity would be $O(1+m(k+1))(n^2)$. Space complexity is based on the size of the adjacency matrix. The size of the adjacency matrix depends on the number of blocks in a subsystem. The space complexity of each level is $O(n^2)$. Complexity analysis for Stateflow model Each state has traversed once and its information is extracted and modeled into a graph. Let the number of states be `S' and transitions be `T'. Then the time complexity is O(S+T). Space complexity is O(S).

Complexity analysis of an SL/SF model having 'm' subsystems in the top level and 'n' Stateflow charts and in each of `m' subsystem there are 'k' other subsystems. The complexity is $O(1+m(k+1))B^2+n(S+T)$. Space complexity for every level is $O(B^2)$.

VI. COMPARISON WITH RELATED WORK

Simulink/Stateflow has originally been designed for the simulation purposes. Automated test generation for Simulink/Stateflow diagram is required to identify the errors. Many authors have tried different ways of test data generation and verification for Simulink/Stateflow diagram.

Zhan [6] proposed one approach, novel search based approach to cover the particular structural elements of Simulink. He has considered the small signal generation as the input, it is not covering all the blocks in the model. But our approach overcoming this limitation our approach covering all the blocks in the model by traversing the model.

Tools also there for the testing of Simulink/Stateflow models, One of these tools is T-Vec Tester [4], generates test cases automatically from the domain testing theory. Another tool is a Reactive Tester [5], by using guided simulations and heuristics without explanation. This approach is limited regarding the length of generating input signals, model size and complexity leads to lower structural coverage. But our approach overcoming these limitations in our approach works for complex models also.

Scaife [7] are able to translate Simulink /Stateflow block into lustre and verify the model using a model checking toll called Lesar. Gadkari et al.[8] have translated Simulink/Stateflow to a formal language and generated test cases based on model checking. Meng Li and Ratnesh kumar introduced a recursive method to translate a Simulink/Stateflow diagram to an Input/Output Extended Finite Automata [9] which is a formal model of reactive untimed infinite state system. In this they generated test cases for the Simulink/Stateflow diagram based on the Input/Output Extended Finite Automata. In our approach we implemented graph from the Simulink/Stateflow diagram and from that we generated test cases.

Mirko Conard et al. [10] proposed one approach to test suite design for code generation tools. They describe the design of a test suite for code generation tools. This method provides solutions of main problems how the correct transformation of a source into a target language can be proved. The application of the proposed testing approach leads to a test suite which is suitable for testing code generators systematically.

The existing code generators can't guarantee that the generated code compiles correctly as mentioned in the design. The reasons are:

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 overcoming these limitations, no need to generate code from the models in our approach because of that it overcome the Mrko Conard's approach. No assumption in our approach so, it overcomes the Reactive Tester approach. We are covering all the blocks and all transitions through the generated graph. The Zhan's approach also not covering all the Blocks due to small signal generation, but our approach overcoming this limitation also.

VII. CONCLUSION AND FUTURE WORK

We proposed a methodology to generate test cases from SL/SF models. First we have constructed 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 many important coverages like state coverage, transition coverage. This is more accurate than the methods which are generating test cases using code generation. We generated test sequences moreover we plan to generate test cases for every Embedded real time control system and we are planning to prioritize test cases based on the certain block.

REFERENCES

[1] The MathWorks, Simulink, Stateflow and Real-TimeWorkshop at http://www.mathworks.com/products, Website, 2003.

[2]Online http://dali.feld.cvut.cz/ucebna/matlab/toolbox/stateow.

[3] B.Beizer. Software Testing Techniques. International Thompson computer press, 1990.

[4] T-VEC, http://www.t-vec.com/.

[5] Reactive Systems Inc, Reactis Simulator / Tester at www.reactive-systems.com, Website 2003.

[6] Y.Zhan. A search-Based Framework for Automatic Test-Set Generation for MATLAB/Simulink Models.PhD thesis, University of York, December 2005.

[7] N.scaife, C.Sofronis, P.Caspi, S.Tripakis, and F. Maraninchi, Defining and translating a safe subset Simulink/stateow into lustre. In EMSOFT 04: Proceedings of the *4th ACM international conferenceon Embedded Soft-ware* .New York, NY, USA:ACM, 2004, pp.259-268.

[8] 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, *4th International Workshop on Model Driven Engineering, Verification and Validation*, 2007.

[9] Meng Li, Ratnesh Kumar Model-Based Automatic Test Generation for Simulink/Stateow using Extended Finite Automaton 2011.

[10] Igno sturmer, Mirko conard "Test suite design for code generation Tools", IEEE 2003.

[11] J Ellson, E Gansner, L Koutso_os, S North, GWoodhull, Graphviz Open Source Graph Drawing Tools, *9th International symposium*, GD 2001 Vienna, Austria, September 23-26, 2001, pages 483-484.