Generation of Test Cases based on Analysis of Simulink Stateflow Models

Basanti Minj
S. o. S. in CS & IT
Pt. R. S. U., Raipur (C. G.)

ABSTRACT

Embedded systems are mainly modeled by using Matlab's Simulink and Stateflow tools. Matlab's Simulink is a tool for modeling, simulating and analyzing software systems and Simulink Stateflow is a control logic tool used to model event-driven systems (Reactive systems) through state machines and flow charts within a Simulink model. In real time, systems undergo frequent changes, thus complexity of the systems grows and testing of the systems become time consuming and expensive even if changes occur in small parts of the system. So, these models need formal verification. In this paper, we focus on event-driven systems which are captured by Simulink Stateflow model. For this, we propose an algorithm *generateFSM* in which we first generate an XML file for the Simulink Stateflow model of a system. Then, we parse that XML file following top-down approach by using an XML parser. Next, we generate a Finite State Machine (FSM) for the model, using the parsed information. By using this FSM, we generate test cases for the models of the embedded systems.

Keywords

Simulink tool, Simulink Stateflow tool, Simulink Stateflow model, Finite State Machine (FSM), Test cases.

1. INTRODUCTION

Every software product undergoes changes during their lifetime. These changes occur due to various reasons such as enhancing functionalities of the existing one, detecting defects in the software product, modification in existing functionalities etc. Every time whenever the changes occur in the software product, the changed software product is to be tested so that the modified code does not negatively affect the behavior of unmodified code. Due to changes, the software product size increases and becomes complex during testing, so the use of appropriate design models for software tasks has become important. Models can be used to represent the desired behavior of a system under test (SUT) or to represent testing strategies and we can test this model through model based testing. Hence, we need formal verification of the models against, stated specifications. Matlab's Simulink/Stateflow (SL/SF) is a software tool for modeling, simulating and analyzing dynamic systems. Simulink is an add-on library for Matlab containing a number of blocks with the help of which one can design dynamic behavior of a system under consideration. To capture discrete control states, one generally uses Stateflow. It provides a graphical editor where we drag Stateflow graphical objects on it from the design palette to create finite state machines. However, since SL/SF doesn't have a published formal semantics, SL/SF model needs to be translated to an intermediate format and from this we can generate intermediate FSM. Using this FSM, we generate test cases for the models of embedded systems.

2. BASIC CONCEPTS

In this section, we discuss the basic concepts required to understand our work.

2.1 Model based testing

It is an application of model based design for designing and also executing artifacts to perform system testing. Here, models are used to represent the desired behavior of a System Under Test (SUT), or to represent testing strategies. This model based testing using models for the generation of system testing procedures. From these models, test cases are derived which are executed against systems under test. Model based testing is very useful for small and large systems. Model based testing has ability to accommodate frequent changes in the requirements.

2.2 Simulink

Simulink is a software tool from the Mathworks for modeling, simulating and analyzing dynamic systems. Most of the embedded and real-time systems that we encounter in real life are hybrid systems. Hybrid systems exhibit both continuous and discrete behavior. These kinds of hybrid systems can be modeled, simulated and analyzed using Simulink. Systems can be modeled in Simulink by creating a network of blocks dragged from the Simulink block library and dropped into the GUI editor and connecting the appropriate port.

2.3 Simulink/Stateflow libraries

- **Source library** - It contains blocks that generate signals.
- **Sink library** - It contains blocks that display output.
- **Discrete library** - It contains blocks that define discrete-time components.
- **Linear library** - It contains blocks that describe linear functions.
- **Connection library** - It contains blocks for implementation of external input/output that passes data to other parts of the model, create subsystems, and perform other functions.
- **Non-linear library** - It contains blocks that describe nonlinear functions.

2.4 Simulink Stateflow

It is an interactive graphical design tool that works with Simulink to capture the event-driven behavior of the systems. Event-driven systems where the system makes a transition from one state to another state based on transition condition. It provides a graphical editor on which the Stateflow graphical objects dragged from the design palette to create finite state machines. Simulink Stateflow enables hierarchical states**.**

States has the following labels:

- **Entry actions** - It define the action to be taken when the state is entered or activated.
- **During actions** - It defines the set of actions to be taken when the state is already active and some event occurs.
- **Exit actions** - It define the actions to be taken when the transition condition become true and the state becomes inactive from active.
- **On event actions** - It defines the actions to be taken when a state is active and the mentioned event occurs.

Transition

Transition in Stateflow means a jump from some source state to any target state.
Transition label consists of the followings:

Event [condition] {condition action} /transition action

- **Event** - It specifies the event that should cause the transition to occur.
- **Condition** - It specifies a boolean expression that needs to be evaluated to true for the transition to take place.
- **Condition action** - It specifies the actions to be immediately executed when the condition evaluates to true.
- **Transition action** - It specifies the action to be executed when the transition destination has been determined to be valid provided the condition is true, if specified.

2.5 Graph visualization software

Graph visualization software (Graphviz) is a package of open source tools initiated by AT and T Labs Research for drawing graphs specified in DOT language scripts. It consists of tools that process DOT files. DOT is a language that describes graphs.

2.6 sourceNode

It is a node that contains the ID of state which is the source state of the transition.

2.7 TransitionNode

It is a node that contains the ID of state which is the destination state of the transition.

2.8 stateEntered

It is a linked list that contains TransitionNode which is already traversed while generating test cases.

3. RELATED WORK

In this section, we discuss some existing related work on test case generation using SL/SF models.

3.1 Automated Translation of MATLAB Simulink/Stateflow Models to an Intermediate Format in HyVisual

This approach [1] specifies the requirements of the intermediate format. Most of the real world Simulink/Stateflow models are hybrid system. They exhibit continuous as well as discrete behavior. So there is an observation lead us to choose an intermediate format to represent these hybrid systems.
The MoBIES team [4] has done considerable research on defining an interchange format for hybrid systems and the result of it is an interchange format called Hybrid System Interchange Format (HSIF). In HSIF, hybrid systems are represented as a network of communicating hybrid automata. In this process huge model can be broken into simple communicating hybrid automata. Hybrid automata communicate with each other through signals and shared variables. If an output signal of hybrid automata A is an input signal of hybrid automata B, then hybrid automata B is said to be data dependent on hybrid automata A.

HyVisual

In HyVisual, They developed a parser for MATLAB model files. The Model Object generated by the parser is used by a Java class called GenMoMLCode which gets the model specific information and generates a HyVisual model represented in Modeling Markup Language (MoML). The generated HyVisual model emulates the Simulink model as a network of hybrid automata. Hence, they choose Network of Hybrid Automata representation as the intermediate format. HyVisual models are represented in an XML based language called Modeling Markup Language (MoML).

3.2 Regression Test Selection Based on Analysis of Simulink/Stateflow Models

The approach [2], presents Simulink/ Stateflow Dependency Graph (SLDG) metamodel. This model comprises of nodes representing different Simulink/Stateflow (SL/SF) model elements along with dependencies capturing the relations between SL/SF elements. They used Model Extractor to parse the mdl file of SL/SF model and generate an intermediate representation of the Simulink Stateflow blocks and the interconnection network of the model named as Simulink/Stateflow Dependency Graph (SLDG).

3.3 A Metamodel for Simulink/Stateflow Models and its Applications

Approach [3], has developed a prototype tool for change impact visualization based on the static analysis of a constructed Simulink/ Stateflow Dependency Graph (SLDG) for the SL / SF model.

3.4 Operational semantics of hybrid systems

Edward and Zheng discuss an interpretation of Hybrid systems as executable models. Hybrid systems are heterogeneous systems that include continuous-time subsystems interacting with discrete events.

In this paper, they focus on the simulation tools, they view that hybrid systems are not much simulated as executed. They view the semantics of hybrid systems as a parallel model of computation and the simulation tools like compilers or interpreters that happen to have a hybrid systems semantics. The executable computational view of hybrid systems was simulated by the DARPA MoBIES (model based integration of embedded software), which begins the challenging task of founding an interchange format for hybrid systems. The aim was to facilitate interchange of models and techniques between tools. The result of this work was a formalism called Hybrid Systems Interchange Format (HSIF).

4. PROPOSED WORK

In this work, we deal with the following issues.

- To decide an intermediate format for representing systems.
- 2. To translate Matlab SL/SF model for the chosen

intermediate format.

- To generate an intermediate Finite State Machine (FSM) by parsing the intermediate format file.
- To generate Test cases from the FSM.

To decide an intermediate format for representing systems we choose an XML file as an intermediate format for representing Matlab SL/SF model. We decide this because Matlab SL/SF model is stored in mdl file which does not give a textual view only graphical view of the model. The XML based language syntax is simple and the model information can be easily retrieved by existing parsers.

To translate Matlab SL/SF model for the chosen intermediate format we use Matlab command to generate XML file for Matlab SL/SF model. We use this XML file as input in our proposed algorithm *generateFSM.*

To generate an intermediate Finite State Machine (FSM) by parsing the intermediate format file we use our proposed algorithm *generateFSM* which take an XML file as an input. The algorithm focus on the Simulink Stateflow part of the SL/SF model which captures the reactive states of the model and generate FSM for it. Then, we use the generated FSM for producing Test Cases.

To generate Test cases from the FSM, we first traverse each *sourceNode* present in linked list containing transition source ID. For each *sourceNode* we find the transition destination id node *TransitionNode*. Then, adding that *TransitionNode* to a new linked list *stateEntered* and we have to add *TransitionNode* to a linked list *stateEntered* till each *TransitionNode* stored in the *stateEntered* linked list.

4.1 Algorithm

In this algorithm, we provide an approach that how we generate test cases by using the Simulink Stateflow model. In this work we have proposed three algorithms, *generateFSM, extract* and *GenerateTestCases*. In *generateFSM* algorithm we parse the XML file (input file) to create Nodelist of transition source node, transition destination node and transition condition node. The *extract* algorithm is used to extract the attributes of the state to maintain its Nodelist. Through these Nodelists we are generating FSM for the XML file of the Simulink Stateflow model. This generated FSM is visualized and validated through an open source tools package named as Graphviz. After this, finally, we are calling another algorithm *GenerateTestCases* for generating test cases of the dynamic systems.

Algorithm 1: generateFSM

```
Input: XML file.
Output: FSM of the stateflow  part of the model.
Variables:
S-list: state node list.
T-list: transition node list.
P-list: P node list.
S-ID: state ID.
S-Name: state label.
T-Name: transition label.
T-ID: transition ID.
SIdList: state ID node list.
SnameList: state labelstring node list.
p-attr: attributes of p.
TconditionList: transition labelstring nodelist.
TIdList: transition ID node list.
Tsource: src tag of transition.
TranSidList: source-state ID list of transitions.
Tdestination: dst tag of transition.
TranDidList: destination-state ID list of transitions.
Tdestination-ID: Transition's destination-state ID node.
Tsource-ID: Transition's source-state ID node.
1: begin
2: parse the XML file to maintain S-list and T-list.
3: for all s ϵ S-list do
4: read elements of s to maintain P-list.
5: for all p ϵ P-list do
6: if p-attr == S-ID then
7: Add S-ID in SIdList.
8: end if
9: if p-attr == S-Name then
10: Add S-Name in SnameList.
11: call extract(S-Name)
12: end if
13: end for
14: end for
15: for all t ϵ T-list do
16: read elements of t to maintain P-list.
17: for all p ϵ P-list do
18: if p-attr == T-Name then
19: Add T-Name in TconditionList
20: end if
21: if p-attr == T-ID then
22: Add T-ID in TIdList
23: end if
24: end for
25: read elements of  t for Tsource
26: for Tsource do
27: read elements of Tsource to maintain P-list
28: for all p ϵ P-list do
29: if p-attr == Tsource-ID then
30: Add Tsource-ID in TranSidList
31: end if
32: end for
33: end for
34: read elements of t for Tdestination
35: for Tdestination do
36: read elements of Tdestination to maintain P-list
37: for all p ϵ P-list do
38: if p-attr == Tdestination-ID then
39: Add Tdestination-ID in TranDidList
40: end if
41: end for
42: end for
43: end for
44: call GenerateTestCases(TranSidList, TranDidList,
TconditionList)
45: Exit
```

Algorithm 2: extract

```
Input: String S-Name
Output: Tokens of state
Variables:
t = " "
t1 = " "
Tokenlist: Token node list.
1: begin
2: for all S-Name do
3: StringTokenizer(S-Name,"delimiter")
4: t = getToken
5: Add t in Tokenlist.
6: while hasMoreTokens do
7: t1 = getToken
8: Add t1 in Tokenlist.
9: end while
10: end for
```

Algorithm 3: GenerateTestCases

```
Input: TranSidList, TranDidList, TconditionList
Output: Test Cases for the Stateflow model.
Variables:
trans = " "
stateEntered: Tdestination-ID node already traversed.
index = 0
1: begin
2: if TranSidList.get(0) == "start" then
3: trans = TranDidList.get(0)
4: add trans in stateEntered
5: index = TranSidList.indexof(trans)
6: if TranSidList.get(index) == trans then
7: repeat
8: perform entry action
9: perform during action
10: until TconditionList.get(index-1) == false
11: get a valid input at S-ID = TranSidList.get(index)
12: if TconditionList.get(index-1) == true then
13: perform exit action
14: perform condition action
15: get a valid input at S-ID = Tran-DidList.get(index)
16: end if
17: end if
18: trans = TranDidList.get(index)
19: if stateEntered.contains(trans) == false then
20: add trans in stateEntered
21: repeat steps 4 to 19
22: else
23: All states are traversed.
24: end if
25: end if
26: Exit
```

4.2 Working of the algorithm

In this subsection we explain our algorithms in a theoretical manner.

Algorithm: generateFSM

In this algorithm we have taken an XML file as input. Here, we parse the XML file for comparing string "state" with the tags in the XML file. If it matches, then we maintain an S-list of tags. Then, for each s ϵ S-list we parse the entire element belong to s for string "P" tag and add to P-list. Then for each p ϵ P-list, we parse all the attributes to get S-Name and S-ID. Then, we create SnameList and SIdList for storing S-Name and S-ID respectively. This process continues till all the s ϵ S-list are parsed.

In the same way we parse the XML file for comparing string "transition" with the tags in the XML file to maintain T-list, which contains transition tags. Then, for each t ϵ T-list we parse the entire element belong to t for string "P" tag, src tag and dst tag. The found P tag must be stored in a P - list. Then for each p ϵ P-list we parse all the attributes to get T-Name and T-ID. Then, we maintain a TconditionList and TIdList for T-Name and T-ID respectively. Then parse elements of src tag for string "P" tag to maintain a P-list. Then for each p ϵ P-list, we parse all the attributes to get a Tsource-ID and add to TranSidList. Then parse the elements of dst tag for string "P" tag to maintain a P-list. Then for each p ϵ P-list we parse all the attributes to get a Tdestination-ID and create a TranDidList. Now all the above lists are used for generating FSM.

Algorithm: extract

In this algorithm, we are taking S-Name as input. Here we use the tokenizer to get tokens of the S-Name to maintain a Tokenlist. This Tokenlist is sent to the calling algorithm.

Algorithm: GenerateTestCases

In this algorithm, we take TranSidList, TranDidList and TconditionList as input. Here we traverse first Tsource-ID in TranSidList and first Tdestination-ID in TranDidList. Then we add Tdestination-ID in *stateEntered*. Then we find the index of Tdestination-ID in TranSidList . We perform an entry action and during action until T-Name in TconditionList is true. If the T-Name is true, then we perform exit action and condition action. Then we get the test case at S-ID = TranSidList.get(index) to reach the Tdestination-ID at TranDidList.get(index) and again we add this Tdestination-ID in *stateEntered*. This process continues till all Tdestination-ID in TranDidList is traversed once. In this way we get test cases for Fan Stateflow model.

5. IMPLEMENTATION

We are explaining our implementation by taking one example of the Simulink Stateflow model.

5.1 Construct Fan Simulink model

It is the model which is drawn in Simulink software tool by dragging Simulink blocks from the Simulink library and dropping into a GUI editor and connecting ports to the blocks for external input and output. This Simulink model also contains chart block which is used to capture reactive systems in the Simulink model. This chart block is in a Stateflow library from which we drag it and drop into a GUI editor.

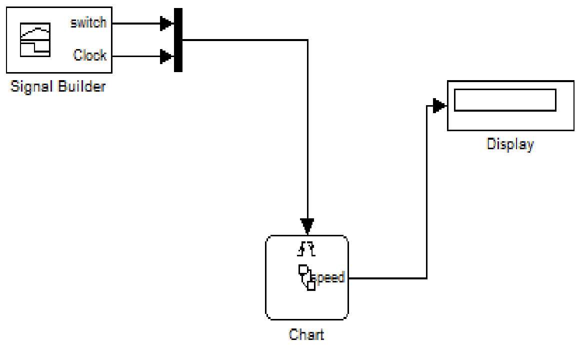


Fig. 1: Fan Simulink model

5.2 Construct Fan Simulink Stateflow model

It is the model which is drawn with the help of interactive graphical design tools that works with Simulink. Here we drag states and transitions from design palette to draw the Stateflow of the reactive systems.

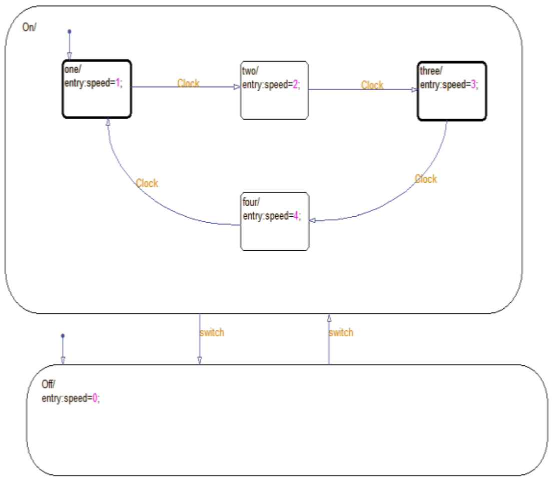
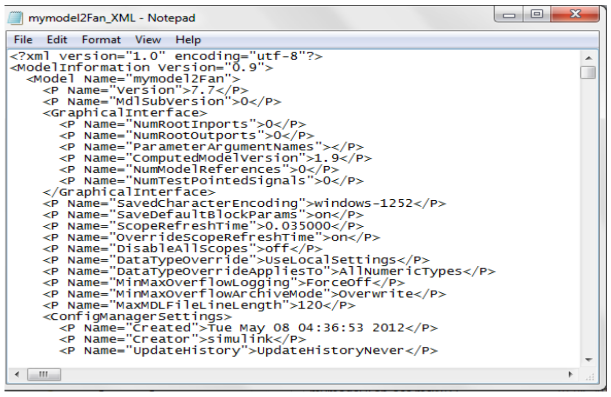


Fig. 2: Fan Simulink Stateflow model

5.3 Generate XML file of the Fan Simulink model

In this we generated XML file of Simulink model. Here we have chosen XML file as an intermediate format so that we have a specification of the Simulink model. Through this specification, we are generating intermediate Finite state machine of the Simulink Stateflow model. The advantage is that XML language is easy to understand thus analyzing it also become easy.



```
<?xml version="1.0" encoding="utf-8"?>
<ModelInformation Version="0.9">
  <Model Name="mymodel2Fan">
    <P Name="Version">7.7</P>
    <P Name="MdlSubVersion">0</P>
    <GraphicalInterface>
      <P Name="NumRootInports">0</P>
      <P Name="NumRootOutports">0</P>
      <P Name="ParameterArgumentNames"></P>
      <P Name="ComputedModelVersion">1.9</P>
      <P Name="NumModelReferences">0</P>
      <P Name="NumTestPointedSignals">0</P>
    </GraphicalInterface>
    <P Name="SavedCharacterEncoding">windows-1252</P>
    <P Name="SaveDefaultBlockParams">on</P>
    <P Name="ScopeRefreshTime">0.035000</P>
    <P Name="OverrideScopeRefreshTime">on</P>
    <P Name="DisableAllScopes">off</P>
    <P Name="DataTypeOverride">UseLocalSettings</P>
    <P Name="DataTypeOverrideAppliesTo">AllNumericTypes</P>
    <P Name="MinMaxOverflowLogging">ForceOff</P>
    <P Name="MinMaxOverflowArchiveMode">Overwrite</P>
    <P Name="MaxMDLFileLineLength">120</P>
    <ConfigManagerSettings>
      <P Name="Created">Tue May 08 04:36:53 2012</P>
      <P Name="Creator">simulink</P>
      <P Name="UpdateHistory">UpdateHistoryNever</P>
```

Fig. 3: XML file of Fan Simulink model

5.4 Generate Finite State Machine for Fan Stateflow model

Here we parse the generated XML file to transform the model into a Finite State Machine (FSM). This finite state machine represents a possible configuration of the systems. From this finite state machine we fine Test cases by searching the executable transitions.

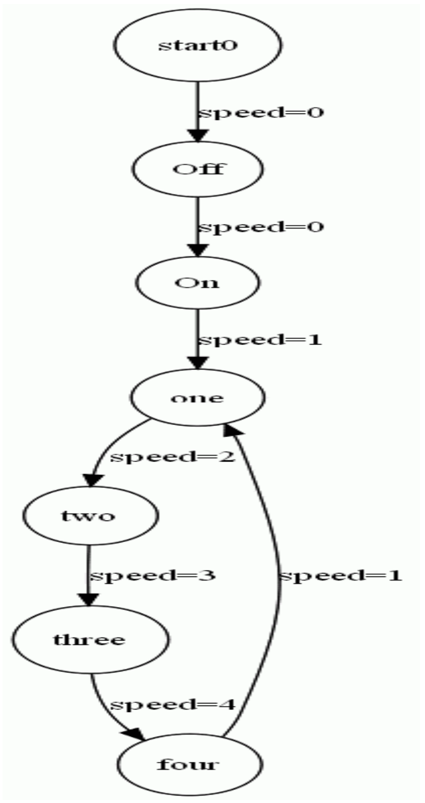
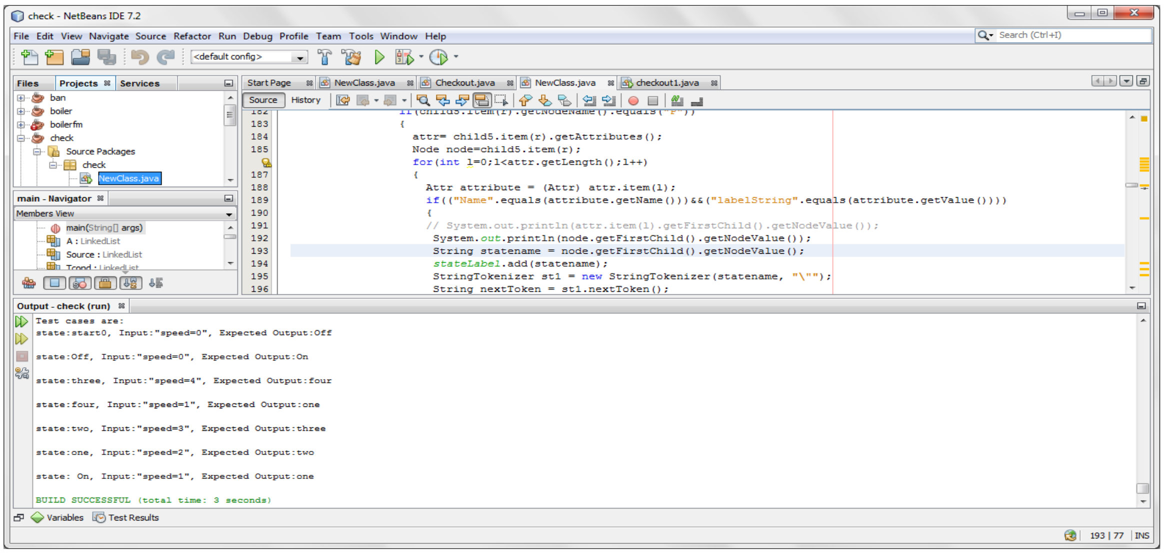


Fig. 4: Finite State Machine for Fan Stateflow model

5.5 Generation of Test Cases for Fan Stateflow Model

Here we generate Test Cases by analyzing finite state machine and searching the executable transitions.



```
if(child5.item(r).getNodeName().equals("F"))
{
  attr= child5.item(r).getAttributes();
  Node node=child5.item(r);
  for(int l=0;l<attr.getLength();l++)
  {
    Attr attribute = (Attr) attr.item(l);
    if(("Name".equals(attribute.getName()))&&("labelString".equals(attribute.getValue())))
    {
    // System.out.println(attr.item(l).getFirstChild().getNodeValue());
     System.out.println(node.getFirstChild().getNodeValue());
     String statename = node.getFirstChild().getNodeValue();
     stateLabel.add(statename);
     StringTokenizer st1 = new StringTokenizer(statename, "\"");
     String nextToken = st1.nextToken();
```

```
Test cases are:
state:start0, Input:"speed=0", Expected Output:Off

state:Off, Input:"speed=0", Expected Output:On

state:three, Input:"speed=4", Expected Output:four

state:four, Input:"speed=1", Expected Output:one

state:two, Input:"speed=3", Expected Output:three

state:one, Input:"speed=2", Expected Output:two

state: On, Input:"speed=1", Expected Output:one

BUILD SUCCESSFUL (total time: 3 seconds)
```

Fig. 5: Test Cases for Fan Stateflow Model

6. RESULT

This table below shows the generated Test Cases in tabular form.

Table 1: Test Cases in tabular form

State	Input	Expected Output
start0	speed=0	Off
Off	speed=0	On
On	speed=1	One
One	speed=2	Two
Two	speed=3	Three
Three	speed=4	Four
Four	speed=1	One

7. COMPARISON

Ray, Rajarshi [1] work on intermediate format of SL/SF model named as hybrid automata represented in XML file. N. Vamshi Vijay [2] work on the intermediate format of a SL/SF model named as Simulink/Stateflow Dependency Graph (SLDG) represented in the mdl file. Nayak, Suraj [3] work on the intermediate format of a SL/SF model named as SLDG represented in mdl file and used a Matlab script for visualization. These comparisons are given in tabular form.

Table 2: comparisons

Paper	Model	Interme-diate format	Intermed-iate file format	Matlab script
Ray et al	Simulink State-flow model	Hybrid auto-mata	XML based language Modelling Markup Language (MoML)	Not used
N. Vamshi Vijay et al	Simu-link State-flow model	Simulink/ State-flow Depend-ency Graph	mdl file	Not used
Nayak et al	Simu-link State-flow model	Simulink/ State-flow Depend-ency Graph	mdl file	Used

8. CONCLUSION

Model based testing is growing more popular in testing area, especially in real time because with the increase in size and complexity of software products an appropriate design models are required for software tasks which can be tested for expected results. Matlab Simulink Stateflow is a software which helps in modelling dynamic systems, but a Simulink Stateflow model may have several levels of hierarchy with several types of implicit dependencies between elements of the model that makes the model complex and difficult to perform any analysis on it. So, the xml file of a model captures all implicit dependencies and represents them explicitly, thus making it possible to perform several types of analysis.

9. REFERENCES

[1] Ray, Rajarshi. “Automated translation of matlab Simulink/Stateflow models to an intermediate format in hyvisual.” Computer Science Department (2007).

[2] N. Vamshi Vijay. “Regression test selection based on analysis of Simulink/stateflow models.” Computer Science Department (2012).

[3] Nayak, Suraj. “A Metamodel for Simulink/Stateflow models and its applications.” Computer Science Department (2013).

[4] Lee, Edward A., and Haiyang Zheng. “Operational semantics of hybrid systems.” Hybrid Systems: Computation and Control. Springer Berlin Heidelberg (2005), pp. 25-53.

[5] Agrawal, Aditya, Gyula Simon, and Gabor Karsai. “Semantic translation of Simulink/Stateflow models to hybrid automata using graph transformations.” Electronic Notes in Theoretical Computer Science 109 (2004), pp. 43-56.

[6] Bringmann, Eckard, and A. Kramer. “Model-based testing of automotive systems.” In Software Testing, Verification, and Validation, (2008) 1st International Conference on, pp. 485-493. IEEE, (2008).

[7] Reicherdt, Robert, and Sabine Glesner. “Slicing MATLAB simulink models.” InSoftware Engineering (ICSE), (2012) 34th International Conference on, pp. 551-561. IEEE, (2012).

[8] Andries, Marc, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. “Graph transformation for specification and programming.” Science of Computer programming 34, no. 1 (1999), pp. 1-54.

[9] Korel, Bogdan, Inderdeep Singh, Luay Tahat, and Boris Vaysburg. “Slicing of state-based models.” In Software Maintenance, (2003). ICSM (2003). Proceedings. International Conference on, pp. 34-43. IEEE, (2003).

[10] Utting, Mark, Alexander Pretschner, and Bruno Legeard. “A taxonomy of model-based testing.” (2006).

[11] MathWorks, “Mathworks MATLAB Simulink.” \\http://www.mathworks.com/products/simulink/