

Generation and Prioritization of Test cases using Simulink/Stateflow models

Basanti Minj



Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, Odisha, India

Generation and Prioritization of Test cases using Simulink/Stateflow models

Thesis submitted in partial fulfilment of the requirements for the degree of

Master of Technology

in

Computer Science and Engineering

(Specialization: Software Engineering)

by

Basanti Minj

(Roll No. 212cs3109)

under the supervision of

Dr. D. P. Mohapatra



Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela, Odisha, 769 008, India

June 2014



Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, Odisha, India.

Certificate

This is to certify that the work in the thesis entitled *Generation and Prioritization of Test cases using Simulink/Stateflow models* by *Basanti Minj* is a record of an original research work carried out by her under my supervision and guidance in partial fulfillment of the requirements for the award of the degree of Master of Technology with the specialization of Software Engineering in the department of Computer Science and Engineering, National Institute of Technology Rourkela. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Place: NIT Rourkela
Date: June 2, 2014

Dr. D. P. Mohapatra
Associate Professor, CSE Department
NIT Rourkela, Odisha

Acknowledgment

I am grateful to numerous local and global peers who have contributed towards shaping this thesis. At the outset, I would like to express my sincere thanks to Dr. D. P. Mohapatra for his advice during my thesis work. As my supervisor, he has constantly encouraged me to remain focused on achieving my goal. His observations and comments helped me to establish the overall direction of the research and to move forward with investigation in depth. He has helped me greatly and been a source of knowledge.

I extend my thanks to our HOD, Dr. S. K. Rath for his valuable advices and encouragement.

My sincere thanks to everyone who has provided me with kind words, a welcome ear, new ideas, useful criticism, or their invaluable time, I am truly indebted.

I must acknowledge the academic resources that I have got from NIT Rourkela. I would like to thank administrative and technical staff members of the Department who have been kind enough to advise and help in their respective roles.

Last, but not the least, I would like to dedicate this thesis to my family, for their love, patience, and understanding.

Basanti Minj
Roll: 212cs3109

Abstract

Embedded systems are mainly modeled by using MATLAB's simulink and stateflow tools. MATLAB's simulink is a tool for modeling, simulating and analysing software systems and 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 stateflow model. For this, we propose an algorithm (*GenerateGraph*) in which we first generate an XML file for the stateflow model of a system. Then, we parse that XML file following top-down approach by using XML parser. Next, we generate intermediate graph for the model, using the parsed information. By using this graph, we generate test cases for the models of the systems having composite states.

Keywords: *MATLAB's Simulink and Stateflow tools; Simulink/Stateflow model; Composite systems; XML file; XML parser;*

Contents

Certificate	i
Acknowledgment	ii
Abstract	iii
List of Figures	vi
List of Tables	viii
1 Introduction	2
1.1 Introduction	2
1.2 Motivation	3
1.3 Basic Concepts	3
1.3.1 Software Testing	3
1.3.2 Simulink	3
1.3.3 Simulink Stateflow	6
1.3.4 Extensible Markup Language (XML)	8
1.3.5 Test case	8
1.3.6 Graph visualization software(GraphViz)	8
1.4 Objective	8
1.5 Problem Statement	9
1.6 Thesis Organization	9
2 Literature Review	11
2.1 Automated Translation of MATLAB Simulink/Stateflow Models to an Intermediate Format in HyVisual	11
2.2 Regression Test Selection Based on Analysis of Simulink/Stateflow Models	12

2.3	A Metamodel for Simulink/Stateflow Models and its Applications . . .	12
2.4	Operational semantics of hybrid systems	12
2.5	Slicing MATLAB Simulink Models	13
2.6	A Dynamic Slicing Technique for UML Architectural Models	13
3	Generation of Test cases using SL/SF	15
3.1	Algorithm	16
3.2	Working of the Algorithm	19
3.3	Implementation	21
3.3.1	implementation steps:	21
3.3.2	Examples	22
3.3.3	Result	32
4	Prioritization of Test cases using SL/SF	35
4.1	Prioritization steps:	35
4.2	Example	36
4.3	Result Analysis	38
5	Conclusion	41

List of Figures

1.1	Inport block	4
1.2	Constant block	4
1.3	Clock block	4
1.4	Outport block	4
1.5	Scope block	5
1.6	Display block	5
1.7	Difference block	5
1.8	Unit Delay block	5
1.9	Integrator block	5
1.10	Gain block	6
1.11	Sum block	6
1.12	Divide block	6
1.13	Switch block	6
3.1	Sample simulink model.	22
3.2	Sample stateflow model.	23
3.3	Generated XML file of sample simulink model.	23
3.4	Generated intermediate graph for sample stateflow model.	24
3.5	Generated Test cases for sample stateflow model.	24
3.6	Fan simulink model.	25
3.7	Fan stateflow model.	25
3.8	Generated XML file for Fan simulink model.	26
3.9	Generated intermediate graph for Fan stateflow model.	26
3.10	Generated Test cases for Fan stateflow model.	27
3.11	Boiler simulink model.	28

3.12	Boiler stateflow model.	28
3.13	<i>On</i> state of Boiler stateflow model.	29
3.14	<i>turn-boiler(mode)</i> state of Boiler stateflow model.	29
3.15	<i>flash-LED()</i> state of Boiler stateflow model.	29
3.16	<i>cold()</i> state of Boiler stateflow model.	30
3.17	Generated XML file of Boiler simulink model.	30
3.18	Generated intermediate graph for Boiler stateflow model.	31
3.19	Generated Test cases for Boiler stateflow model.	31

List of Tables

3.1	Test cases for sample stateflow model	32
3.2	Test cases for Fan stateflow model	33
3.3	Test cases for Boiler stateflow model	33
4.1	Test cases for Boiler stateflow model	36
4.2	Computing IF value for each State	37
4.3	Prioritized Test cases for Boiler Stateflow model	38
4.4	Faults detected by non prioritized test cases	39
4.5	Faults detected by prioritized test cases	39

Chapter 1

Introduction

Introduction

Motivation

Basic Concepts

Problem Statement

Chapter 1

Introduction

1.1 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 of a system represents the needed behavior of the system or to represent an approach for testing and we can test this model through model based testing. Hence, we need formal verification of the models against, stated specifications.

MATLAB's Simulink software tool helps in modeling the systems, analyzing dynamic systems and simulating the systems. A Simulink library containing various blocks by using these blocks we can design required behavior of a system under consideration. To capture reactive states of a system, we use Stateflow in Simulink. Stateflow provides an editor where we drag objects on editor from the design palette to create Stateflow of the reactive systems. However, since the Simulink model doesn't have a textual view of the formal semantics, Simulink model needs to be translated to an intermediate textual representation and from this we can generate intermediate graph. Using this graph, we generate test cases

for the models of composite systems.

1.2 Motivation

MATLAB Simulink/Stateflow is one of the widely used industrial tools. It helps in modeling systems, even if they are more complex. The resulting model must be tested in order to detect faults in the systems. But, such model consists of a large number of blocks, due to which the testing process becomes complex. So we have to decrease the complexity of the models to handle large models and to ensure the quality of the complex models.

1.3 Basic Concepts

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

1.3.1 Software Testing

It is needed to investigate whether the system under test meeting the desired behavior and works as expected. Software Testing depends on the testing method employed.

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.

1.3.2 Simulink

Simulink is a software tool provided by the Mathworks from which we can model, simulate and analyze dynamic systems. Many embedded systems present in

real life are hybrid systems. Hybrid system consists of both continuous and discrete nature. So, these systems can be modeled, simulated and analyzed using Simulink. Systems can be modeled in Simulink by dragging blocks from the Simulink block library and dropped into the GUI editor and connecting the appropriate ports with the blocks.

Simulink/Stateflow libraries

1. **Source library** - It contains blocks that generate signals.

Example:

Inport block - It is the block which shows input going to the subsystem.

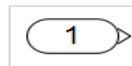


Figure 1.1: Inport block

Constant block - This block is used to generate constant value.



Figure 1.2: Constant block

Clock block - This block displays the simulation time.



Figure 1.3: Clock block

2. **Sink library** - It contains blocks that display output.

Example:

Outport block - It is the block which shows output coming from the subsystem.

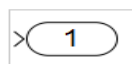


Figure 1.4: Outport block

Scope block - It is the block which is used to display the signals.



Figure 1.5: Scope block

Display block - This block is used to Show the value of input.



Figure 1.6: Display block

3. **Discrete library** - It contains blocks that define discrete-time components.

Example:

Difference block - This block shows the output after subtracting the current input value from the previous input value.

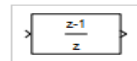


Figure 1.7: Difference block

Unit delay block - This block holds and delays its input by one sample period.

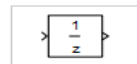


Figure 1.8: Unit Delay block

4. **Continuous library** - It contains blocks that describe linear functions.

Example:

Integrator block - This block integrates its input signal with respect to time.

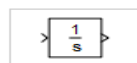


Figure 1.9: Integrator block

5. **Math Operations library** - This block performs mathematical calculations.

Example:

Gain block - It multiplies its input signal by a constant value (gain).

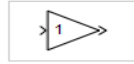


Figure 1.10: Gain block

Sum block - This block is used to perform addition or subtraction on its inputs.



Figure 1.11: Sum block

Divide block - This block divides its first input by its second input and shows the output.

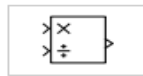


Figure 1.12: Divide block

6. **Non-linear library** - It contains blocks that describe nonlinear functions.

Example:

Switch block - The Switch block passes through the first input or the third input based on the value of the second input taking as Threshold parameters.

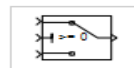


Figure 1.13: Switch block

1.3.3 Simulink Stateflow

It is a stateflow design tool from which we can draw a stateflow of the systems and this stateflow design tool works with Simulink to capture the event-driven behavior of the systems. Event-driven systems where system makes a transition from one state to another state based on transition condition. It provides an editor

on which the stateflow objects dragged from the design palette to create stateflow of the systems. Simulink Stateflow enables hierarchical states.

States has labels

Entry actions- It defines the action to be carried out 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 defines the actions to be taken when the transition condition become true and the state becomes inactive from active.

Transition

Transitions in Stateflow means a jump from some source state to some target state. Transition label consists:

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 action 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.

1.3.4 Extensible Markup Language (XML)

It is a markup language that defines the set of rules that is used for encoding documents in a format that is readable by human and machine. It is a format with the strong support of the Unicode Standard. The Unicode Standard consists of an encoding method, set of standard character encodings, set of code charts for viewable reference, etc.

- An intermediate representation of SL/SF model is an XML file that captures all implicit and explicit dependencies.
- XML language syntax is simple and the model information can be easily retrieved by the use of existing XML parsers.

SAX (Simple API for XML)

It is an event-based parser API for XML . SAX parsers read each piece of the data from an XML document sequentially. SAX parser works as a stream parser and it is unidirectional. The event includes XML element node, XML processing instruction, etc. These events are fired when encounters by the XML parser.

1.3.5 Test case

A test case is a set of conditions under which whether one of software system features is working as it was originally established for it to do.

1.3.6 Graph visualization software(GraphViz)

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.

1.4 Objective

- To generate test cases for systems having composite states.

- To Prioritize the generated test cases.

1.5 Problem Statement

This thesis work focuses on generating test cases for composite systems problem. First we draw SL/SF model for composite systems using MATLAB's Simulink and Stateflow graphical design tool . From SL/SF model we generate XML specification of the model. Then, we draw intermediate graph through XML specification. Then, we generate test cases by traversing each node of the generated graph. Finally, we prioritized the generated test cases.

1.6 Thesis Organization

Organization of thesis is as follows: Chapter 2 describes literature review. Chapter 3 describes generation of test cases. Chapter 4 describes prioritization of test cases. Finally, Chapter 5 concludes the thesis work.

Chapter 2

Literature Review

Chapter 2

Literature Review

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

This approach [1], specifies the requirements of the intermediate format in HyVisual. In HyVisual model, they choose a network of Hybrid Automata representation as the intermediate format. HyVisual models are represented in an XML based language called Modeling Markup Language (MoML).

This paper also discusses the translator implementation. The MATLAB, Simulink models are saved as .mdl files that contains all necessary information related to the blocks and the connections present in the model which are needed for simulation and visualization of the model in MATLAB. For the required information about the Simulink models parser is needed for parsing the .mdl file. They implemented a parser in Java using Jdk 1.6 and Jflex 1.4.1. The parser generates model Object that is used by a Java class called GenMoMLCode that generates a HyVisual model represented in MoML. The generated HyVisual model is a network of hybrid automata.

Advantages:

- Handling automata of a model is easier than the model itself.
- XML language is simple to understand and the model information can be easily recovered from existing XML parsers.

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

This 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).

2.3 A Metamodel for Simulink/Stateflow Models and its Applications

This 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.

2.4 Operational semantics of hybrid systems

This paper [4], consider an interpretation of Hybrid systems as executable models. Hybrid systems consist of continuous-time subsystems combined with discrete events.

In this paper, they focus on the simulation tools, they view that hybrid systems are not much simulated as executed. 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 intention was to provide an interchange of models and techniques between tools. The output was a Hybrid Systems Interchange Format (HSIF).

2.5 Slicing MATLAB Simulink Models

This paper [5], presents a static slicing method for MATLAB, Simulink models using dependence graph based approach. They contributed to-

- Analyze the control and data dependencies present in the Simulink models.
- A slicing approach for Simulink model that holds all the semantics and hierarchy of the model.

In order to lower the complexity of a model by removing those parts that do not affect the control and data dependence.

2.6 A Dynamic Slicing Technique for UML Architectural Models

This paper [6], presents a dynamic program slicing techniques to split big architectures into small convenient portions. They used UML architectural models for which they prepare an intermediate representation of the model named as a Model Dependency Graph (MDG) that captures all existing dependencies between model elements. Then, for any given slicing criteria they traverse the MDG to find out the significant parts from an architectural model based on the dependencies between them to figure out a dynamic slicing of the architectural model.

Chapter 3

Generation of Test cases using SL/SF

Algorithm

Working of the Algorithm

Implementation

Chapter 3

Generation of Test cases using SL/SF

In our work, we carried out the followings.

1. To construct an intermediate representation for representing systems under consideration.
2. To translate the simulink model to the chosen intermediate representation.
3. To generate the intermediate graph by parsing XML file of simulink model.
4. To generate test cases from the intermediate graph.

To construct an intermediate representation for representing systems we choose an XML file as an intermediate format for representing simulink model. We decide this because simulink model is stored in .mdl file which does not give a textual view only graphical view of the model. An XML language is simple to understand so that model information are easily recovered by existing parsers.

To translate the simulink model to the chosen intermediate format we use MATLAB command to generate XML file for simulink model. Then, we use this XML file as an input in our proposed algorithm GenerateGraph.

To generate an intermediate graph by parsing XML file, we use our proposed algorithm GenerateGraph which take an XML file as an input. Our algorithm focus on stateflow part of the simulink model which captures the reactive states of

the model and generate intermediate graph for it. Then, we use generated graph for generating Test cases.

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

3.1 Algorithm

Here, we describe our algorithm step by step:

Algorithm: GenerateGraph

Input: XML file of simulink model

Output: Graph of the stateflow part of the simulink model.

step 1: Draw simulink model by using MATLAB simulink tool and stateflow model is added to the simulink model by using MATLAB stateflow design tool.

Step 2: Generate XML file for the simulink model.

Step 3: Parse the generated XML file for stateflow states and transitions node list.

step 4: Find a node list of source state and destination state of each transition.

Step 5: Generate an intermediate graph by using node list of source state and destination state of transitions.

Step 6: Generate the test cases by using intermediate graph.

In this algorithm, we provide an approach that how we generate test cases by using simulink models. Our algorithm name is GenerateGraph within this algorithm two sub algorithm are there named as Extract (S-Name) and GenerateTestCases. In GenerateGraph algorithm we parse the XML file (input file) to

create nodelist of transition source node, transition destination node and transition condition node. This GenerateGraph algorithm has sub algorithm called Extract (S-Name) is used to extract the state attributes nodelist. Through these nodelists we are generating intermediate graph from that XML file of the simulink model. This generated intermediate graph is visualized and validated through an open source tools package named as Graphviz. By visualizing this intermediate graph and by sub algorithm GenerateTestCases we are generating test cases for the dynamic systems.

Algorithm 1 GenerateGraph

Input: XML file.
Output: Graph 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 node list.
 TIdList: transition ID node list.
 Tsource: src tag of transition.
 TranSIdList: source-state ID list of transition.
 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 ∈ S-list do
4:   read elements of s to maintain P-list.
5:   for all p ∈ 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 ∈ T-list do
16:   read elements of t to maintain P-list.
17:   for all p ∈ 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 ∈ 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 ∈ 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 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 valid input at S-ID = TranDidList.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**

3.2 Working of the Algorithm

In this subsection we explain our algorithms in theoretical manner.

Algorithm: GenerateGraph

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 $s \in$ S-list we parse all the element belong to s for string “P” tag and add to P-list. Then for each $p \in$ 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 $s \in S$ -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 $t \in T$ -list we parse all the 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 $p \in P$ -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 $p \in P$ -list, we parse all the attributes to get 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 $p \in P$ -list we parse all the attributes to get Tdestination-ID and create a TranDidList. Now all the above lists are used for generating intermediate graph.

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 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 stateflow model.

3.3 Implementation

We are explaining our implementation step by step:

3.3.1 implementation steps:

1. Construct 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.

2. Construct stateflow model

It is the model which is drawn on the chart block present in the simulink model. This model is drawn with the help of a stateflow design tool. This design tool works with the simulink model because chart block is used only in a simulink model. Here we drag states and transitions from the design palette of the design tool to draw the stateflow of the reactive systems.

3. Generate XML file of 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 graph of the stateflow part of the simulink model. The advantage is that XML language is easy to understand thus analyzing it also become easy.

4. Generate intermediate graph for stateflow model

Here we parse the generated XML file to transform the stateflow part of the

model into an intermediate graph. This intermediate graph represents possible configuration of the reactive systems. From this graph we find Test cases.

5. Generation of Test cases for stateflow model

Here we generate Test cases by analyzing intermediate graph and searching the executable transitions.

3.3.2 Examples

We are explaining our implementation by taking some Simulink/Stateflow models.

I. Sample of simulink/stateflow model that is a simple model.

1. Construct sample simulink model

It is the sample simulink model which is drawn with the help of simulink software tool by dragging a chart block from stateflow library and dropped into it. This chart block is used to capture reactive systems in the sample simulink model.

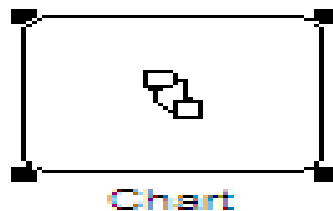


Figure 3.1: Sample simulink model.

2. Construct sample stateflow model

This model is drawn in the chart block by using the stateflow design tool. Here we draw the sample stateflow model of the reactive systems. There are A , B and C states. Where, each state contains an entry, during and exit action as required.

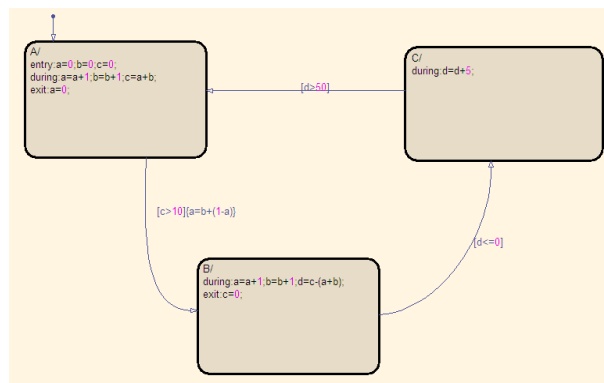


Figure 3.2: Sample stateflow model.

3. Generate XML file for sample simulink model

Here, we generated XML file of the sample simulink model. By using this file specification, we are generating intermediate graph for the sample stateflow of the model.

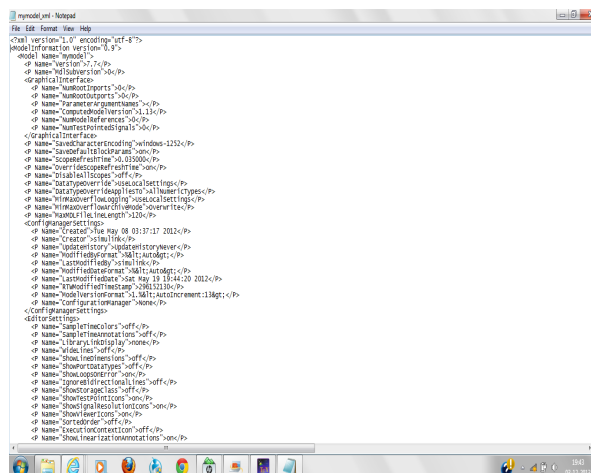


Figure 3.3: Generated XML file of sample simulink model.

4. Generate intermediate graph for sample stateflow model

Here we parse the generated XML file to transform the sample stateflow model into an intermediate graph to represent the possible design of the reactive systems of the sample stateflow model.

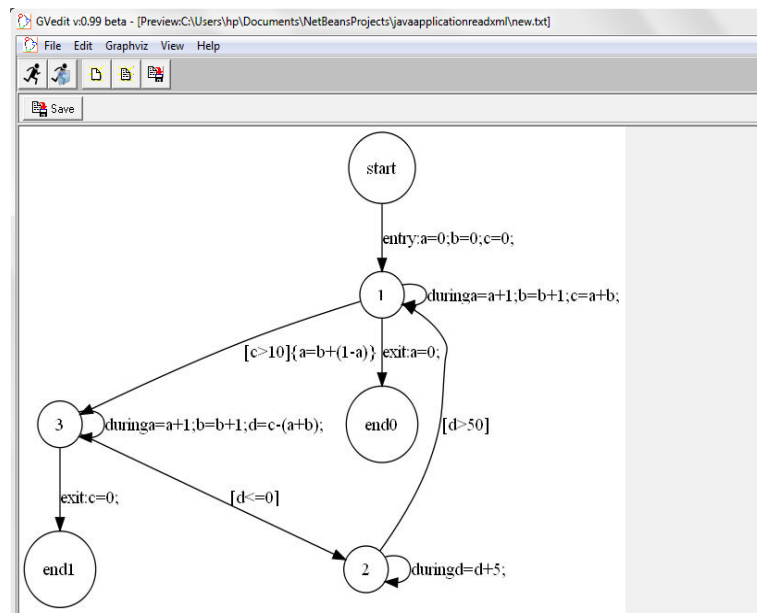


Figure 3.4: Generated intermediate graph for sample stateflow model.

5. Generation of Test cases for sample Stateflow model

Here we generate Test cases by analyzing intermediate graph and searching the executable transitions.

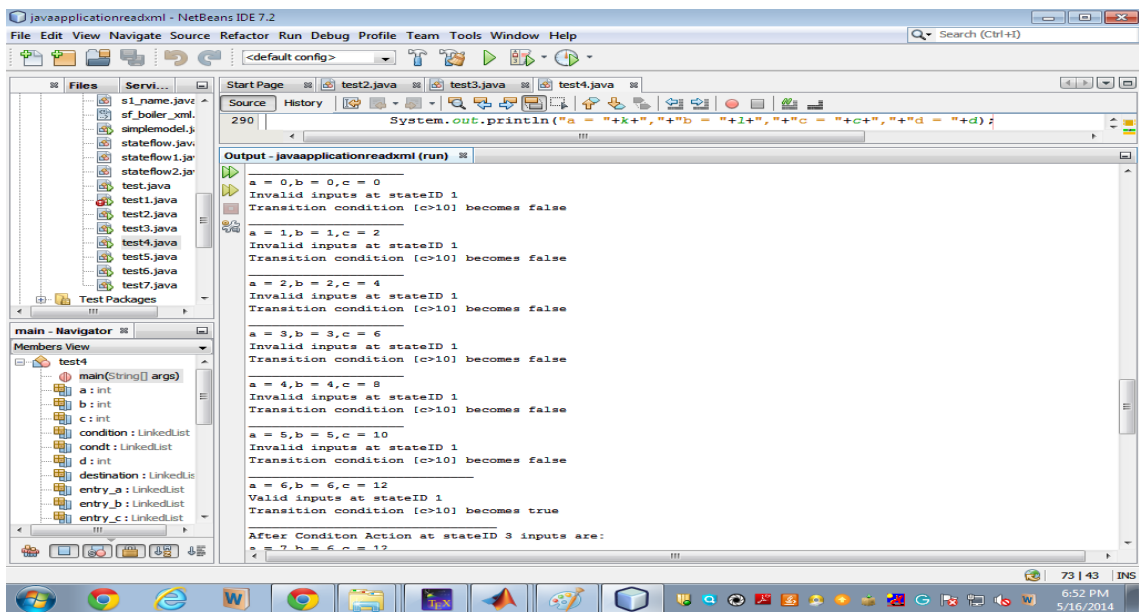


Figure 3.5: Generated Test cases for sample stateflow model.

II. Simulink model of a composite object- Fan

1. Construct Fan simulink model

It is the Fan simulink model which is drawn with the help of simulink software tool by dragging simulink blocks such as signal Builder, Display blocks from the simulink library and chart block from stateflow library which are dropped into a GUI editor and connecting ports to the blocks for external input and output. This chart block is used to capture reactive systems in the Fan simulink model.

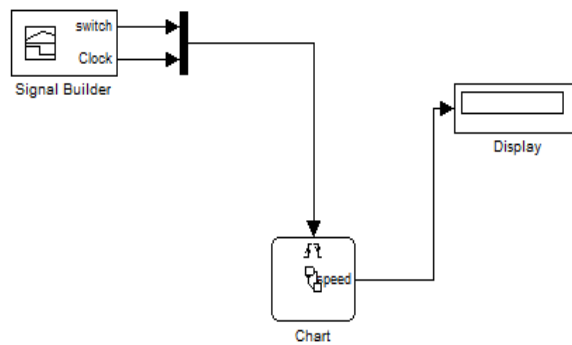


Figure 3.6: Fan simulink model.

2. Construct Fan stateflow model

This Fan stateflow model is drawn on the chart block present in Fan simulink model with the help of a stateflow design tool. Here we drag states and transitions from design palette to draw the stateflow of the reactive systems. In the Fan stateflow model, there are *Off* and *On* states where, *On* is a composite state because it contains child states named as *one*, *two*, *three* and *four* working as regulator of Fan.

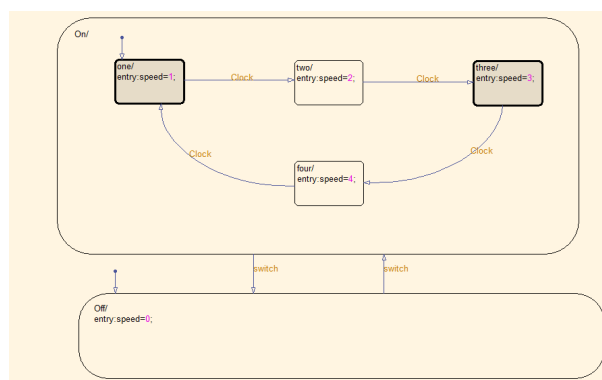
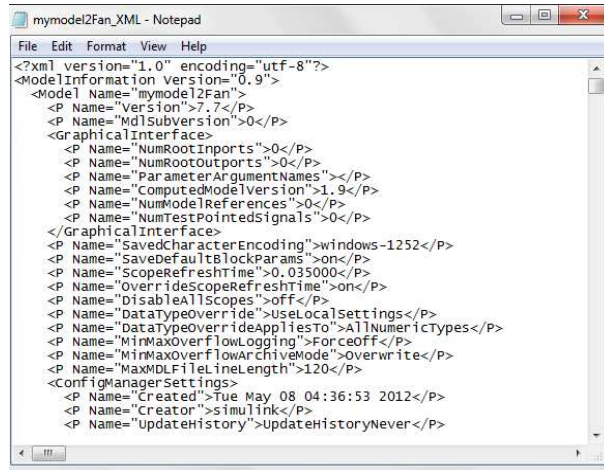


Figure 3.7: Fan stateflow model.

3. Generate XML file for Fan simulink model

In this we generated XML file of the Fan simulink model. By using this file specification, we are generating intermediate graph for the Fan stateflow part of the simulink model.



```

mymodel2Fan_XML - Notepad
File Edit Format View Help
<?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="NumRootInputs">0</P>
      <P Name="NumRootOutputs">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="MaxMdlFileInLength">120</P>
    <ConfigManagerSettings>
      <P Name="Created">Tue May 08 04:36:53 2012</P>
      <P Name="Creator">simulink</P>
      <P Name="UpdateHistory">UpdateHistoryNever</P>
    </ConfigManagerSettings>
  </Model>
</ModelInformation>

```

Figure 3.8: Generated XML file for Fan simulink model.

4. Generate intermediate graph for Fan stateflow model

Here we parse the generated XML file to transform the Fan stateflow model into an intermediate graph. This intermediate graph represents possible configuration of the Fan reactive systems.

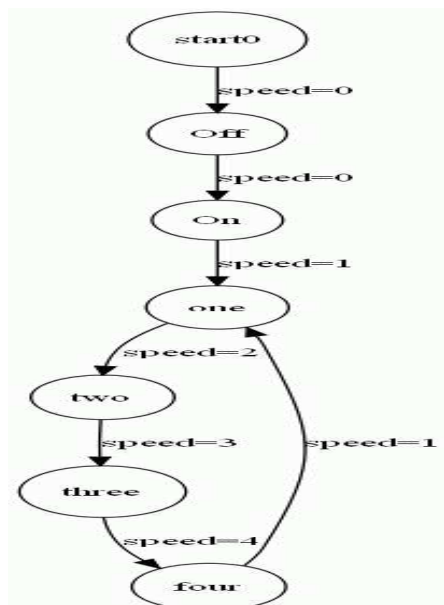


Figure 3.9: Generated intermediate graph for Fan stateflow model.

5. Generation of Test cases for Fan stateflow model

Here we generate Test cases by analyzing intermediate graph of the Fan reactive systems and searching the executable transitions.

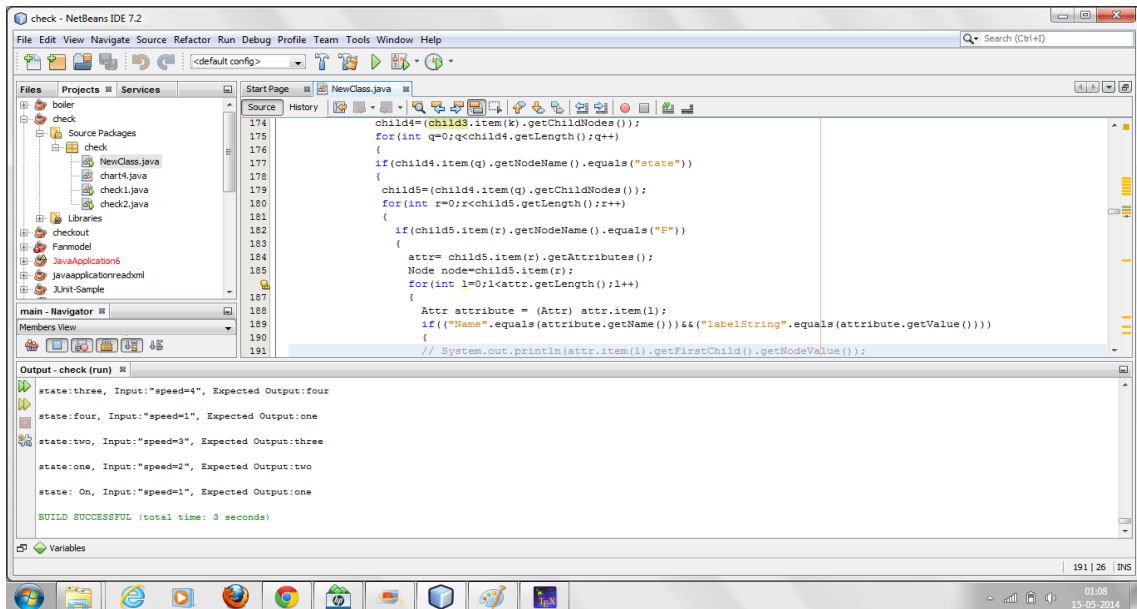


Figure 3.10: Generated Test cases for Fan stateflow model.

III. Simulink model of a composite object- Boiler.

1. Construct Boiler simulink model

It is the Boiler simulink model which is drawn with the help of simulink software tool by dragging simulink blocks such as Constant, SubSystem, Scope blocks, etc. from the simulink library and chart block from stateflow library which are dropped into a GUI editor and connecting ports to the blocks for external input and output. This chart block is used to capture reactive systems of the Boiler simulink model.

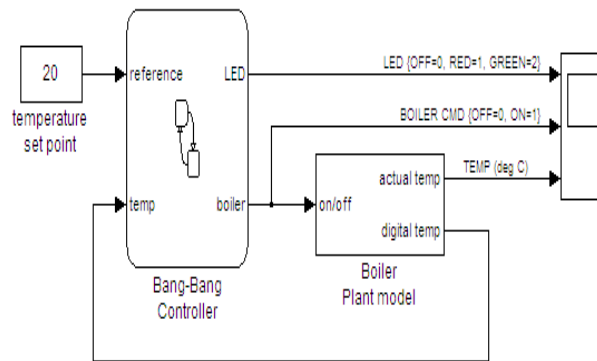


Figure 3.11: Boiler simulink model.

2. Construct Boiler stateflow model

It is the model which is drawn on the chart block present in the Boiler simulink model with the help of a stateflow design tool. Here we drag states and transitions from design palette to draw the stateflow of the Boiler reactive systems. In the Boiler stateflow model, *Heater* state contains *Off*, *On*, *Flash* state where, *Heater* state contains *cold()* function state, *Off* state contains *turn-boiler(OFF)* function state, *Flash* state contains *flash-LED()* function state.

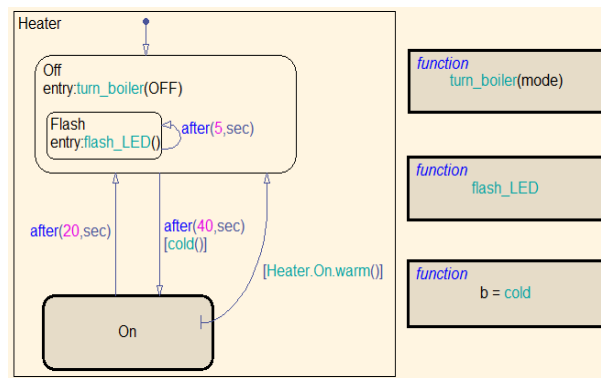
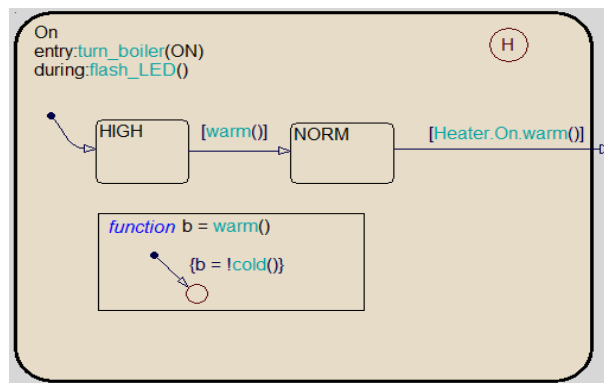
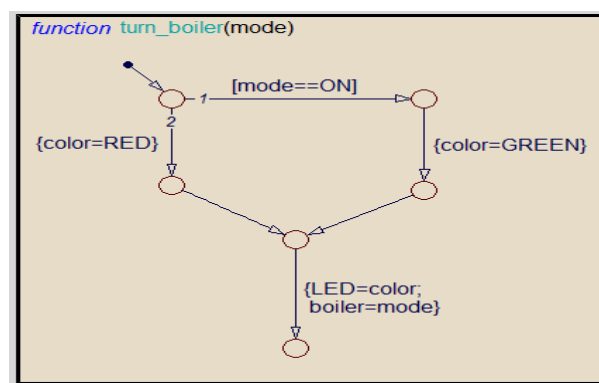


Figure 3.12: Boiler stateflow model.

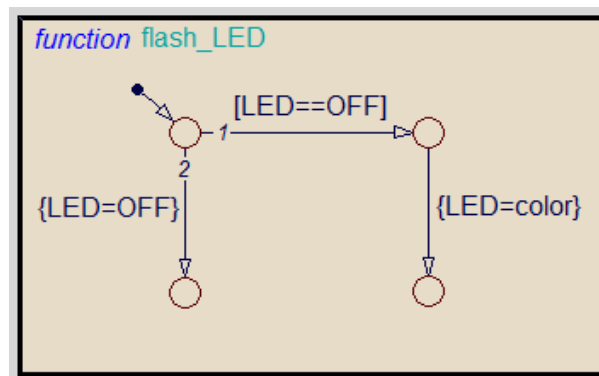
In this, *On* state is a composite state because it contains child states named as *HIGH*, *NORM* and function state named as *turn-boiler(ON)*, *flash-LED()* and *warm()*.

Figure 3.13: *On* state of Boiler stateflow model.

This *turn-boiler(mode)* function state contains junctions and transitions to perform this particular function.

Figure 3.14: *turn-boiler(mode)* state of Boiler stateflow model.

This *flash-LED()* function state contains junctions and transitions to perform the color changing operation of the LED.

Figure 3.15: *flash-LED()* state of Boiler stateflow model.

This *cold()* function state contains junction and a default transition to compare the input temperature with the given reference temperature of the Boiler.

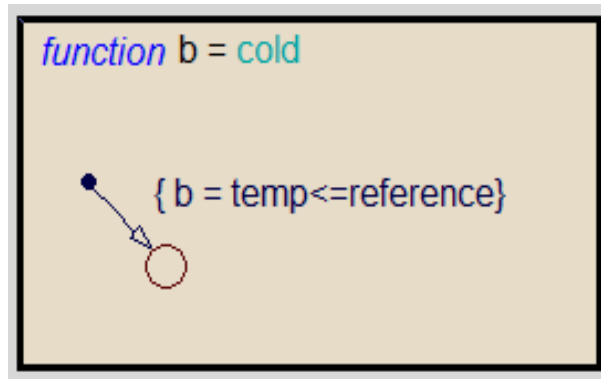


Figure 3.16: *cold()* state of Boiler stateflow model.

3. Generate XML file for Boiler simulink model

In this we generated XML file of the Boiler simulink model. By using this file specification, we are generating intermediate graph for the Boiler stateflow part of the Simulink model.

```

sf_boiler_xml - Notepad
File Edit Format View Help
<?xml version="1.0" encoding="utf-8"?>
<ModelInformation Version="0.9">
  <Model Name="sf_boiler">
    <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">1.89</P>
      <P Name="NumModelReferences">0</P>
      <P Name="NumTestPointedSignals">0</P>
    </GraphicalInterface>
    <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">UseLocalSettings</P>
    <P Name="MinMaxOverflowArchiveMode">overwrite</P>
    <P Name="MaxMdlFileLength">120</P>
  </Model>
  <ConfigManagerSettings>
    <P Name="Created">Wed May 10 03:26:16 2000</P>
    <P Name="Creator">The Mathworks Inc.</P>
    <P Name="UpdateHistory">updateHistoryNever</P>
  </ConfigManagerSettings>
</ModelInformation>

```

Figure 3.17: Generated XML file of Boiler simulink model.

4. Generate intermediate graph for Boiler stateflow model

Here we parse the generated XML file to transform the Boiler stateflow model into an intermediate graph. This intermediate graph represents possible configuration of the Boiler reactive systems.

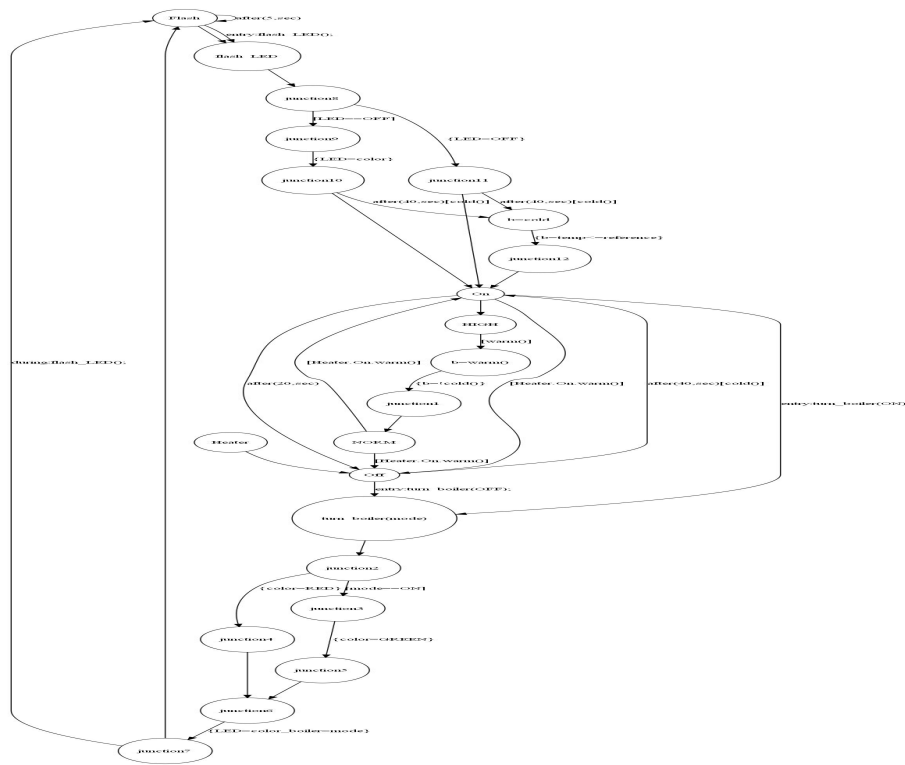


Figure 3.18: Generated intermediate graph for Boiler stateflow model.

5. Generation of Test cases for Boiler stateflow model

Here we generate Test cases by analyzing intermediate graph of the Boiler reactive systems and searching the executable transitions.

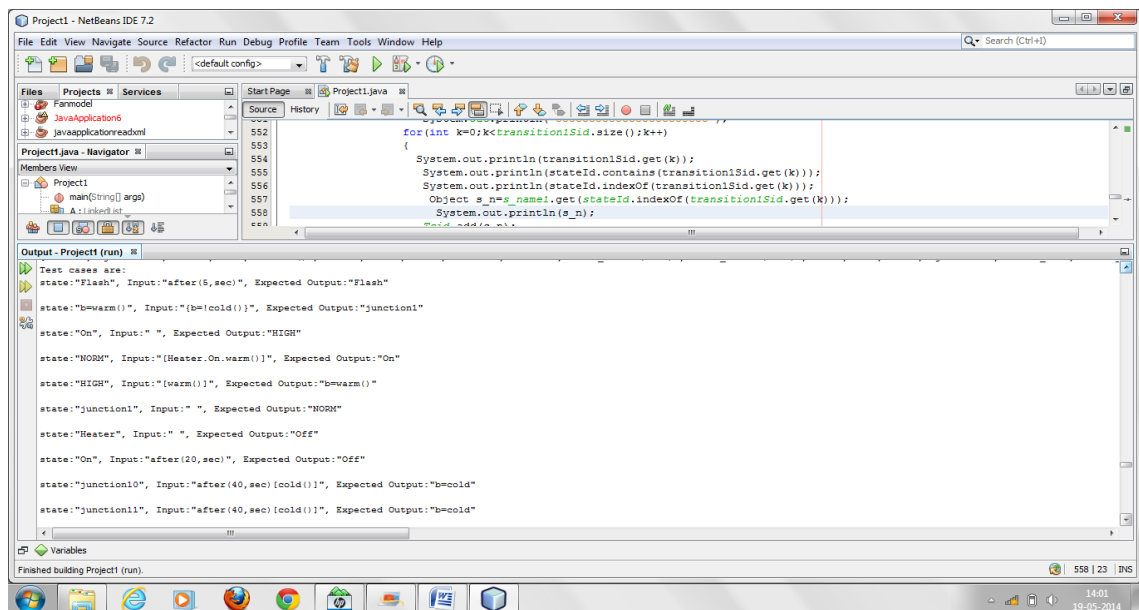


Figure 3.19: Generated Test cases for Boiler stateflow model.

3.3.3 Result

In this, we represent our generated Test cases for the stateflow models in tabular form.

Table 3.1 shows the Test cases for sample stateflow model.

Table 3.1: Test cases for sample stateflow model

State id	Input	Condition	Exit action	Condition action	Expected State id
	$a=a+1, b=b+1, c=a+b$	$[c \leq 10]$			
1	$a=0, b=0, c=0$	False			1
1	$a=1, b=1, c=2$	False			1
1	$a=2, b=2, c=4$	False			1
1	$a=3, b=3, c=6$	False			1
1	$a=4, b=4, c=8$	False			1
1	$a=5, b=5, c=10$	False			1
1	$a=6, b=6, c=12$	True	$a=0$	$a=b+(1-a)$	3
	$a=a+1, b=b+1, d=c-(a+b)$	$[d \leq 0]$			
3	$a=7, b=6, c=12, d=-1$	True	$c=0$		2
	$d=d+5$	$[d \leq 50]$			
2	$a=7, b=6, c=0, d=4$	False			2
2	$a=7, b=6, c=0, d=9$	False			2
2	$a=7, b=6, c=0, d=14$	False			2
2	$a=7, b=6, c=0, d=19$	False			2
2	$a=7, b=6, c=0, d=24$	False			2
2	$a=7, b=6, c=0, d=29$	False			2
2	$a=7, b=6, c=0, d=34$	False			2
2	$a=7, b=6, c=0, d=39$	False			2
2	$a=7, b=6, c=0, d=44$	False			2
2	$a=7, b=6, c=0, d=49$	False			2
2	$a=7, b=6, c=0, d=54$	True			1

Table 3.2 shows the Test cases for Fan stateflow model.

Table 3.2: Test cases for Fan stateflow model

TC id	State	Input	Expected Output
1	start0	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=1	one

Table 3.3 shows the Test Cases for Boiler Stateflow model.

Table 3.3: Test cases for Boiler stateflow model

TC id	State	Input	Expected Output
1	Flash	after(5,sec)	Flash
2	b=warm()	b=!cold()	junction1
3	NORM	[Heater.On.warm()]	On
4	HIGH	[warm()]	b=warm()
5	On	after(20,sec)	Off
6	junction10	after(40,sec)[cold()]	b=cold
7	junction11	after(40,sec)[cold()]	b=cold
8	On	entry:turn-boiler(ON);	turn-boiler(mode)
9	Off	entry:turn-boiler(OFF);	turn-boiler(mode)
10	On	[Heater.On.warm()]	Off
11	Off	after(40,sec)[cold()]	On
12	NORM	[Heater.On.warm()]	Off
13	Flash	entry:flash-LED();	flash-LED
14	junction7	during:flash-LED();	Flash
15	junction2	[mode==ON]	junction3
16	junction2	color=RED	junction4
17	junction3	color=GREEN	junction5
18	junction6	LED=color-boiler=mode	junction7
19	junction8	[LED==OFF]	junction9
20	junction8	LED=OFF	junction11
21	junction9	LED=color	junction10
22	b=cold	b=tempj=reference	junction12

Chapter 4

Prioritization of Test cases using SL/SF

Prioritization steps

Example

Result Analysis

Chapter 4

Prioritization of Test cases using SL/SF

4.1 Prioritization steps:

1. Construct simulink model.
2. Generate XML file for the simulink model.
3. Generate intermediate graph using XML file.
4. Compute Fan_in for each state present in an intermediate graph.
5. Compute Fan_out for each state present in an intermediate graph.
6. Compute product of Fan_in and Fan_out for finding information flow (IF) value of each state.
7. State having higher IF value present in test case is prioritized first.

To compute Fan_in for each state we have to compare each state with all the transition destination nodes if state matches than we increases the count and this counting continues till one iteration of transition destination nodes completed and we store the count value in Fan_in of a state. This process continues till we compute Fan_in for all state.

To compute Fan_out for each state we have to compare each state with all the transition source nodes if state matches than we increases the count and this counting continues till one iteration of transition source nodes completed and we store the count value in Fan_out of a state. This process continues till we compute

Fan_out for all state.

For finding IF value of each state we are computing product of Fan_in and Fan_out of each state.

$$IF(A) = Fan_in(A) * Fan_out(A) \quad (4.1)$$

where, $Fan_in(A)$ - Number of states calling state A.

$Fan_out(A)$ - Number of states called by state A.

$IF(A)$ - Information flow value of state A.

State with higher IF value represents that the state having higher complexity so the test cases are prioritized based on the higher IF value of the transition source state.

4.2 Example

Considering, generated test cases for Boiler model.

Table 4.1: Test cases for Boiler stateflow model

TC id	State	Input	Expected Output
1	Flash	after(5,sec)	Flash
2	b=warm()	b=!cold()	junction1
3	NORM	[Heater.On.warm()]	On
4	HIGH	[warm()]	b=warm()
5	On	after(20,sec)	Off
6	junction10	after(40,sec)[cold()]	b=cold
7	junction11	after(40,sec)[cold()]	b=cold
8	On	entry:turn-boiler(ON);	turn-boiler(mode)
9	Off	entry:turn-boiler(OFF);	turn-boiler(mode)
10	On	[Heater.On.warm()]	Off
11	Off	after(40,sec)[cold()]	On
12	NORM	[Heater.On.warm()]	Off
13	Flash	entry:flash-LED();	flash-LED
14	junction7	during:flash-LED();	Flash
15	junction2	[mode==ON]	junction3

16	junction2	color=RED	junction4
17	junction3	color=GREEN	junction5
18	junction6	LED=color- boiler=mode	junction7
19	junction8	[LED==OFF]	junction9
20	junction8	LED=OFF	junction11
21	junction9	LED=color	junction10
22	b=cold	b=temp<=reference	junction12

In table 4.2, we are computing Fan_in and Fan_out of each states to find IF value of each states.

Table 4.2: Computing IF value for each State

State	Fan_in	Fan_out	IF=Fan_in* Fan_out
Heater	0	1	0
Off	4	2	8
Flash	3	3	9
On	5	4	20
HIGH	1	1	1
NORM	1	2	2
b=warm()	1	1	1
junction1	1	1	1
turn_boiler(mode)	2	1	2
junction2	1	2	2
junction3	1	1	1
junction4	1	1	1
junction5	1	1	1
junction6	2	1	2
junction7	1	2	2
flash_LED	2	1	2
junction8	1	2	2
junction9	1	1	1
junction10	1	2	2
junction11	1	2	2
b=cold	2	1	2
junction12	1	1	1

In table 4.3, we are prioritizing generated test cases based on the higher IF value of source states.

Table 4.3: Prioritized Test cases for Boiler Stateflow model

TC id	State	Input	Expected Output
1	On	after(20,sec)	Off
2	On	entry:turn-boiler(ON);	turn-boiler(mode)
3	On	[Heater.On.warm()]	Off
4	Flash	after(5,sec)	Flash
5	Flash	entry:flash-LED();	flash-LED
6	Off	entry:turn-boiler(OFF);	turn-boiler(mode)
7	Off	after(40,sec)[cold()]	On
8	NORM	[Heater.On.warm()]	On
9	junction10	after(40,sec)[cold()]	b=cold
10	junction11	after(40,sec)[cold()]	b=cold
11	NORM	[Heater.On.warm()]	Off
12	junction7	during:flash-LED();	Flash
13	junction2	[mode==ON]	junction3
14	junction2	color=RED	junction4
15	junction6	LED=color-boiler=mode	junction7
16	junction8	[LED==OFF]	junction9
17	junction8	LED=OFF	junction11
18	b=cold	b=temp<=reference	junction12
19	b=warm()	b=!cold()	junction1
20	HIGH	[warm()]	b=warm()
21	junction3	color=GREEN	junction5
22	junction9	LED=color	junction10

4.3 Result Analysis

Computing Average Percentage of Fault Detected (APFD) for non prioritized test cases and prioritized test cases.

$$APFD = 1 - \sum_{k=1}^m Pos(F_k)/nm + 1/2n \quad (4.2)$$

where, n - number of test cases.

m - number of faults.

Pos(Fk) - the position of the first test case revealing the fault Fk.

Table 4.4: Faults detected by non prioritized test cases

TC/ Fau-																						
Its	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
F1					*																	
F2														*								
F3			*	*						*		*										
F4						*	*															*
F5								*	*						*							
F6													*	*					*			

APFD for non prioritized test cases:

$$APFD = 1 - (5 + 14 + 3 + 6 + 8 + 13) / (22 * 6) + 1 / (2 * 22) \quad APFD = 0.65$$

Table 4.5: Faults detected by prioritized test cases

TC/ Fau-																						
Its	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
F1	*																					
F2												*										
F3			*					*			*									*		
F4									*	*									*			
F5		*				*							*									
F6					*							*					*					

APFD for prioritized test cases:

$$APFD = 1 - (1 + 12 + 3 + 9 + 2 + 5) / (22 * 6) + 1 / (2 * 22) \quad APFD = 0.78$$

Chapter 5

Conclusion

Chapter 5

Conclusion

Model based testing is growing more popular in the testing area, especially in real time because as the size of software products increases the complexity is also increasing. Therefore, an appropriate design model is required for software tasks which can be tested for expected results. MATLAB's simulink and stateflow is a software which helps in modelling dynamic systems, but a simulink 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.

Bibliography

- [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.”, M.Tech. thesis, IIT kharagpur, Computer Science Department (2012).
- [3] Suraj Nayak. “A Metamodel for Simulink/Stateflow models and its applications.”, M.Tech. thesis, IIT kharagpur, 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] Reicherdt, Robert, and Sabine Glesner. “Slicing MATLAB simulink models.” Software Engineering (ICSE), (2012) 34th International Conference on, pp. 551-561. IEEE, (2012).
- [6] Lallchandani, Jaiprakash T., and Mall, Rajib. “A dynamic slicing technique for UML architectural models.” Software Engineering, IEEE Transactions on 37, no. 6 (2011), pp. 737-771.
- [7] Sabharwal, Sangeeta, Ritu Sibal, and Chayanika Sharm. “Applying Genetic Algorithm for Prioritization of Test Case Scenarios Derived from UML Diagrams.” International Journal of Computer Science Issues (IJCSI) 8, no. 3 (2011).

-
- [8] Kavitha, R., and N. Sureshkumar. “Test Case Prioritization for Regression Testing based on Severity of Fault.” *International Journal on Computer Science & Engineering* 2, no. 5 (2010).
- [9] 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.
- [10] 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).
- [11] Andries, Marc, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jrg Kreowski, Sabine Kuske, Detlef Plump, Andy Schrr, and Gabriele Taentzer. “Graph transformation for specification and programming.” *Science of Computer programming* 34, no. 1 (1999), pp. 1-54.
- [12] 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).
- [13] Mund, G. B., and Mall, Rajib. “An efficient interprocedural dynamic slicing method.” *Journal of Systems and Software* 79, no. 6 (2006), pp. 791-806.
- [14] Bates, Samuel, and Horwitz, Susan. “Incremental program testing using program dependence graphs.” In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, (1993), pp. 384-396.
- [15] Liang, Donglin, and Harrold, Mary Jean. “Slicing objects using system dependence graphs.” In *Software Maintenance, 1998. Proceedings.*, International Conference on, pp. 358-367. IEEE, (1998).

-
- [16] Korel, Bogdan, Singh Inderdeep, Tahat Luay , and Vaysburg Boris. “Slicing of state-based models.” In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pp. 34-43. IEEE,(2003).
- [17] Chen, Yanping, Robert L. Probert, and Ural Hasan. “Model-based regression test suite generation using dependence analysis.” In *Proceedings of the 3rd international workshop on Advances in model-based testing*, pp. 54-62. ACM, (2007).
- [18] Bringmann, Eckard, and Kramer Andreas. “Model-based testing of automotive systems.” In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pp. 485-493. IEEE, (2008).
- [19] Lee, David, and Yannakakis Mihalis. “Principles and methods of testing finite state machines-a survey.” *Proceedings of the IEEE* 84, no. 8 (1996), pp. 1090-1123.
- [20] Dalal, Siddhartha R., Ashish Jain, Nachimuthu Karunanithi, J. M. Leaton, Christopher M. Lott, Gardner C. Patton, and Horowitz, Bruce M. “Model-based testing in practice.” In *Proceedings of the 21st international conference on Software engineering*, pp. 285-294. ACM, (1999).
- [21] Utting, Mark, Alexander Pretschner, and Bruno Legeard. “A taxonomy of model-based testing.” (2006).
- [22] MathWorks, “Mathworks MATLAB Simulink.”
<http://www.mathworks.com/products/simulink/>