Generating Test Sequences and Slices for Simulink/Stateflow Models

Adepu Sridhar
(Roll no.: 211CS3301)

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

Generating Test Sequences and Slices for Simulink/Stateflow Models

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

Master of Technology

by

Adepu Sridhar

Roll no-211CS3301

under the guidance of

Dr. Durga Prasad Mohapatra

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

May 2013

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

June 3, 2013

CERTIFICATE

This is to certify that the thesis titled ***"Generating Test Sequences and Slices for Simulink/Stateflow Models "*** submitted by ***Adepu Sridhar***, in partial fulfillment for the award of the Master of Technology in Computer Science and Engineering from the Department of Computer Science, National Institute of Technology Rourkela, India is a bonafide record of work carried out by him under my supervision and guidance. The Thesis has fulfilled all the requirements as per the regulations of this institute and, in my opinion, has reached the standard needed for submission.

Prof. Durga Prasad Mohapatra
Associate Professor

ACKNOWLEDGEMENT

With grateful heart I thank God for giving me knowledge, wisdom, good health and willingness to work sincerely throughout the tenure of my project work.

Then with great pleasure and deep sense of gratitude, I take this opportunity to express my sense of indebtedness to Prof. Durga Prasad Mohapatra, Dept. of Comp. Sc and Engg, NIT Rourkela, for erudite guidance, affectionate encouragement, wholehearted involvement in my project without which it would have been difficult for me to complete this work. I have worked with him for more than a year now and he inspired me both as a human being and as an advisor. I am also gratefully thankful to Prof. Korra sathyababu, Dept of Comp. Sc and Engg, NIT Rourkela, for his continuous support.

I express my gratitude to Prof. Rajib Mall of IIT Kharagpur for providing the necessary inputs and guidance at different stages of my work.

Most importantly, none of this would have been possible without the love of my Parents, Brother and Sister. My family to whom this dissertation is dedicated to, has been a constant source of love, concern, support and strength all these years. I would like to express my heartfelt gratitude to them.

I would like to thank all my friends and lab-mates for their encouragement and understanding. Their help can never be penned with words.

Adepu Sridhar

Abstract

In a typical software development project more than 50 percent of software development effort is spent in testing phase. Test case design as well as execution consumes a lot of time. So automated generation of test cases is highly required. In our thesis we generated test sequences from Simulink/Stateflow, which is used to develop Embedded control systems. Embedded Control Systems are now integral parts of many application systems in the areas of Aerospace, Communication, Automobiles, etc. Simulink/Stateflow (SL/SF) is being used for the development of reactive systems and Embedded Control Systems.

Testing of these systems is very important in order to provide error free systems as well as quality assurance. For these purpose Test cases are used to test the systems. We developed the test sequences which are use to generate test cases. First, we represent the System using Simulink/Stateflow models. For this purpose normally we use Simulink tool, which is available in the MATLAB. We developed the dependency graph from the SL/SF model. For Simulink part of the model we use Out put dependency and for the Stateflow part of the model we use Control dependency graph. From those graphs we generate the test sequences. These test sequences are used to used to generate test cases.

MATLAB Simulink/Stateflow is one of the widely used industrial tools for developing complex embedded systems in the automotive sector. The resulting Simulink/Stateflow models often consist of more than ten thousand blocks and a large number of hierarchical levels. To ensure the quality of such models, automated static analyses and slicing methods are necessary to cope up with this complexity. In particular, static analyses are required that operate directly on the models. In this, we present an approach for slicing Simulink/Stateflow models using dependence graphs from the automotive and avionics domain. With slicing, the complexity of a model can be reduced to a given point of interest by removing unrelated model elements, thus paving the way for subsequent static quality assurance methods.

Keywords: Simulink/Stateflow, Dependency Graph, Test Sequences, Test case, Slicing

Contents

1 Introduction **2**
1.1 Simulink/Stateflow Models(SL/SF) 2
1.2 Motivation 3
1.3 Organization of our thesis 4

2 Basic Concepts **5**
2.1 Basics of Testing 5
2.1.1 Types Of Testing 6
2.1.2 Types of system testing 8
2.2 Model Based development 9
2.2.1 Model Based Design Languages 10
2.3 Simulink 11
2.4 Stateflow 12
2.5 Basic concepts of Slicing 13
2.6 Objective 16

3 Review of Related Work **17**
3.1 Literature review on testing of Simulink Models 17
3.2 Literature review of Slicing of Simulink Models 21

4 Generating Test sequences for Simulink/Stateflow models **23**
4.1 Generating Test sequences for Simulink/Stateflow models 23
4.2 Implementation Details 27
4.3 Tools used for the implementation 27
4.4 Implementation Results 29
4.4.1 An illustartive Example 30
4.4.2 Complexity Analysis 34

5 Slicing of Simulnik/Stateflow models 36
5.1 Dependencies in SL/SF models 36
5.2 Slicing of Simulink/Stateflow models 38
5.2.1 Computing conditional execution contexts 38
5.2.2 Building the dependence graph 38
5.2.3 Computing the Simulink/Stateflow Slice 39
5.3 Slicing Example 39

6 Conclusions and Future Work 43

BIBLIOGRAPHY 46

List of Figures

2.1 SDLC and V Model . . . 7
2.2 Static and Dynamic Testing . . . 8
2.3 Notations in the Stateflow . . . 14
2.4 An Example Program and its CFG . . . 15

4.1 Block Diagram of our Approach . . . 24
4.2 Stateflow of fan model . . . 30
4.3 Top level graph . . . 31
4.4 Secondary level graph . . . 32
4.5 Example of Simulink model . . . 33
4.6 Graph of Simulink model . . . 33

5.1 An Example Simulink/Stateflow model . . . 39
5.2 Stateflow part of the Model . . . 40
5.3 The model dependence graph for the SL/SF model in Figure 5.1 and Figure 5.2 . . . 41
5.4 Marked model dependence graph showing the static slice w.r.t. the slicing criterion ‘mul’ . . . 42

Chapter 1

Introduction

Quality assurance is an important issue for software development. Even though many quality assurance techniques, such as design reviews, formal specifications, model checking, and inspections are available today, testing is still the primary means for achieving quality at industrial and government sites.

While the employment of systematic design and development practices results in increasingly reliable software, some errors are still likely to be present in the software. The goal of testing [1] is to expose hidden errors by exercising the software on a set of test cases. In its simplest form, a test case consists of program inputs and corresponding expected outputs. After testing of the total software, we get the confidence on it.

Software testing is very labor intensive and hence also expensive. It can account for 50 percent of the total cost of software development. Therefore tools that automate one or more aspects of testing can greatly help in managing the the overall cost of testing. Testing techniques can be broadly classified into two categories, functional and structural. Functional testing [2] is concerned with functionality rather than implementation of the program. Therefore it involves exercising different input output conditions. Structural testing is concerned is with testing the implementation of the program by exercising different programming structures used by the program.The primary focus of this report is model based testing.

1.1 Simulink/Stateflow Models(SL/SF)

Embedded Control Systems are now integral parts of many application systems in the areas of Aerospace, Communication, Automobiles, etc. As a result, scientists and engineers are

looking for easy and reliable techniques to design, develop, test and verify these systems. With model based design and development becoming a trend, industries use design and simulation tool sets like MATLAB and Mathematica.

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

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

1.2 Motivation

Use of Simulink/Stateflow in industries like auto motives, especially in flight control software for the next manned space missions. That software has to be reliable. The developers of the flight software [3] use model-based design using MathWorks with UML 2.0.

In the past two decades the avionics software community has increasingly applied model-based software engineering, where models are used to specify software designs, and often executable code is generated automatically from the models. The models are expressed in domain-specific modeling languages with higher-level abstractions that are well known and convenient for domain engineers. Flight control software have been developed using Simulink/Stateflow.

The systems developed using Simulink/Stateflow in Flight control software in NASA

are big and complex and safety critical. As a result of which, the number of test cases required to test SL/SF models is usually very large. Test case execution is both automatic and manual. However, huge amount of effort, resources and time is required to execute the test cases manually. As per a report from NASA it took 27 days to execute 135 test cases for an embedded telecoms software of size 300 KLOC. It is clear from this fact that test for SL/SF models is a very expensive activity.

The model must be tested in order to detect faults in the Simulink/Stateflow model as early as possible. Exhaustive test is not possible for any system. Testing of those systems are very costly. Flight control software is usually based on various states and for that states representation often uses Stateflow diagrams utilization of the internal structure of the diagram to generate Test cases is important. This is achieved by covering Stateflow coverage and particular states coverage.

1.3 Organization of our thesis

Chapter 2 In this chapter we discuss the basic concepts, and definitions used in the rest of the thesis.

Chapter 3 Provides the literature review of test case generation for Simulink/Stateflow models and few approaches of slicing.

Chapter 4 Provides the basic approach for generating test sequences of SL/SF models and the tools that we have used and Case study.

Chapter 5 Discusses the approach of slicing, and Implementation with example.

Chapter 6 In this Chapter we are Concluding the thesis and discusses the future work.

Chapter 2

Basic Concepts

This chapter provides some background concepts and the definitions of some basic terms that we will use in the rest of the thesis. First of all, we will explain the fundamentals of testing. Then we discuss Simulink/Stateflow models. Then we discuss the procedure to generate test sequences for Simulink models and to generate the coverage report of stateflow chart of a simulink model. Then we discuss the fundamentals of testing and regression test selection. Next, we define control flow graph, system dependence graph and state flow graph. Finally we will discuss and define control dependency and data dependency for both control flow graphs and state flow graphs.

2.1 Basics of Testing

Testing a program consists of subjecting the program to a set of test inputs (or test cases) and observing if the program behaves as expected. The aim is to identify all defects existing in software product. It provides a practical way of reducing defects in a system and increasing the user's confidence in the development system. Software testing is performed to reveal possible failures of the software. During testing, a software product is executed with test cases, and the output is observed to detect possible failures.

Software testing is one of the verification and validation software practices.

Verification: Are we building the product right? Through verification, we make sure the product behaves the way we want it to. Validation: Are we building the right product? Through validation, we check to make sure that somewhere in the process a mistake hasn't been made such that the product build is not what the customer asked for; validation always involves comparison against requirements.

The following terms are responsible for how the incorrect result might occur:

- Mistake - A human action that produces an incorrect result.
- Fault or Defect - An incorrect step, process or data definition in a program.
- Fault or Defect An incorrect step, process or data definition in a program.
- Error - The difference between a computed, observed, or measured value or Condition and the true, specified, or theoretically correct value or condition.

2.1.1 Types Of Testing

In this section, we discuss the different types of testing [1]:

Black Box Testing: Without considering the internal part of the code, we test the software. In this Block Box Testing we test the software by considering the requirements specification.

Disadvantages of Black Box Testing:

- Cannot be directed toward specific segments of code which may be very complex (and therefore more error prone).
- Most testing related research has been directed toward glass box testing.

White Box Testing: It deals with the internal structure of the code. White box testing is also called as glass, structural, open box or clear box testing. In order to implement white box testing, the tester has to deal with the code and hence is needed to possess knowledge of coding and logic i.e. internal working of the code. White box test also needs the tester to look into the code and nd out which unit/statement/chunk of the code is malfunctioning.

Advantages of white box testing are

- The other advantage of white box testing is that it helps in optimizing the code.
- It helps in removing the extra lines of code, which can bring in hidden defects.

Advantages of Gray Box Testing:

- Combination of the advantages of black box testing and white box testing.

Disadvantages of Gray Box Testing

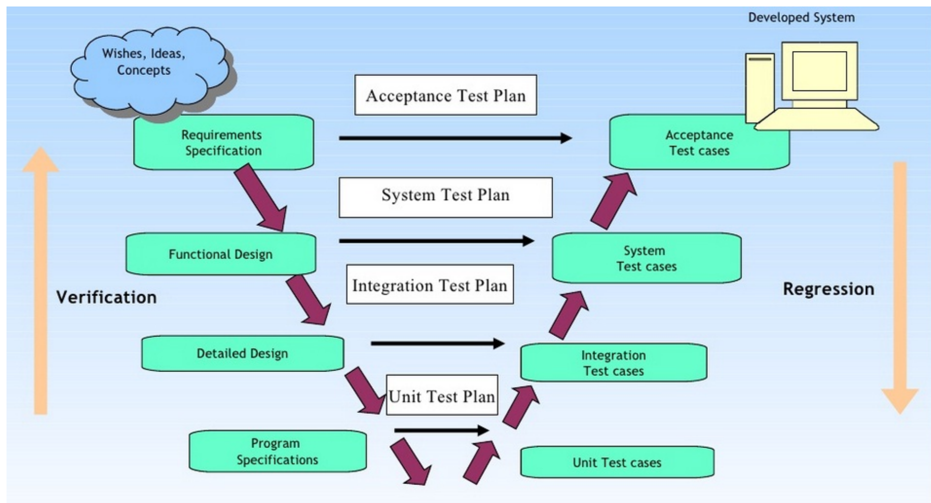


Figure 2.1: SDLC and V Model

- Combination of the disadvantages of black box testing and white box testing.

Testing can be divided into static and dynamic testing [1] based on the execution of the code. In Dynamic testing we execute the code and static testing done without executing code.

Static Testing: Code reviews, inspections and walk throughs are static testing. These are verification activites. Code Reviews, inspection and walkthroughs are few of the static testing methodologies.

Dynamic Testing: In dynamic testing the software must actually be compiled and run. Dynamic testing is the validation portion of Verification and Validation. These are the Validation activities.

There are several levels [2] of testing that should be done on a large software system. Each level of test has a specification that defines the correct behavior the test is examining so that incorrect behavior (an observed failure) can be identified. The six levels and the origin of specification involved in the test levels are now discussed.

Unit Testing: Unit testing is the testing of individual hardware or software units or groups of related units. Unit testing is generally done within a class or a component.

Integration Testing: Integration test is testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them. Using both black and white box testing techniques, the tester verifies that units work together when they are integrated into a larger code base. Just because the components

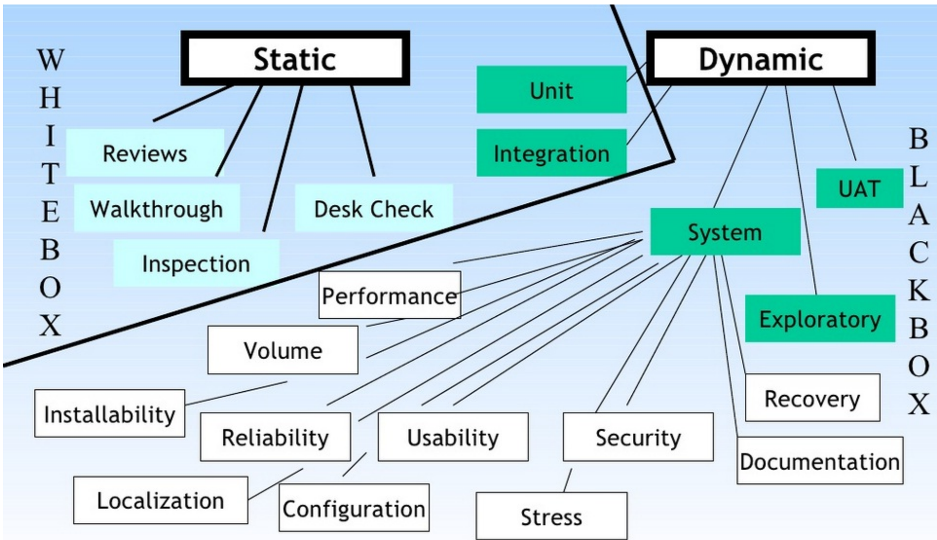


Figure 2.2: Static and Dynamic Testing

work individually, that doesn't mean that they all work together when assembled or integrated. To plan these integration test cases, testers look high-level and low-level design documents.

Functional and System Testing: Functional testing involves ensuring that the functionality specified in the requirement specification works. System testing involves putting the new program in many different environments to ensure the program works in typical customer environments with various versions and types of operating systems and/or applications. Because system test is done with a full system implementation and environment, several classes of testing can be done that can examine nonfunctional properties of the system. It is the best when function and system testing is done by an unbiased, independent perspective.

2.1.2 Types of system testing

- Performance- performance testing is in general testing performed to determine how a system performs in terms of responsiveness and stability under a particular workload.

- Stress-Stress testing (torture testing) is a form of deliberately intense or thorough testing used to determine the stability of a given system or entity. Reasons can include: - to determine breaking points or safe usage limits; to confirm intended specifications are being met; to determine modes of failure, and to test stable operation of a part or

system outside standard usage.

- Usability- It measures the user friendly of the system.
- Recovery- It deals when crashes or power failure in those cases, how to recover and any data loss.
- Configuration-Rather than testing for performance from the perspective of load, tests are created to determine the effects of configuration changes to the system's components on the system's performance and behaviour.
- Compatibility- It deals whether,The software is possible to run in different environments.
- Reliability-Software reliability testing is a field of testing in which deals with testing a software's ability to function under given environmental conditions for a particular amount of time. Once these problems are discovered and corrected, it provides assurance that the software is in it's best possible condition.

Acceptance Testing: After functional and system testing, the product is delivered to a customer and the customer runs black box acceptance tests based on their expectations of the functionality. In this customer checks himself whether the software satisfying his requirements.

Regression Testing: Throughout all testing cycles, regression test cases are run. Regression testing is selective re-testing of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements.

Beta Testing: When an advanced partial or full version of a software package is available, the development organization can offer it free to one or more (and sometimes thousands) potential users or beta testers. These users install the software and use it as they wish, with the understanding that they will report any errors revealed during usage back to the development organization. These users are usually chosen because they are experienced users of prior versions or competitive products.

2.2 Model Based development

Model-Based Design (MBD) [4] is a mathematical and visual method of addressing problems associated with designing complex control, signal processing and communication systems.

It is used in many motion control, industrial equipment, aerospace, and automotive applications. Model-based design is a methodology applied in designing embedded software. MBD provides an efficient approach for establishing a common framework for communication throughout the design process while supporting the development cycle ("V" diagram). In model-based design of control systems, development is manifested in these four steps:

1. modeling a plant
2. analyzing and synthesizing a controller for the plant
3. simulating the plant and controller
4. integrating all these phases by deploying the controller.

The model-based design [5] paradigm is significantly different from traditional design methodology.

2.2.1 Model Based Design Languages

- Formal languages
- High level programming languages (C, C++)
- Finite State Machines, Statecharts, SpecCharts, Stateflow
- Block structured language: Data Flow Diagrams, Lustre, Signal, Simulink
- Concurrent Languages (CSP, SDL)
- Hardware description languages (VHDL, Verilog)
- Visual language Unified Modeling Language(UML)

In computer programming, a block is a section of code which is grouped together. Blocks consist of one or more declarations and statements. A programming language that permits the creation of blocks, including blocks nested within other blocks, is called a block structured programming language. Algo 58 is the first Block structure language.

In the model based development Simulink is a tool to design the model. These are programs that runs as a companion to MATLAB, these programs are developed and marketed by the MathWorks.

2.3 Simulink

Simulink/Stateflow (SL/SF) [6] is being used for the development of reactive systems quite heavily both in industry and academia. Embedded Control Systems are now integral parts of many application systems in the areas of Aerospace, Communication, Automobiles, etc. As a result, scientists and engineers are looking for easy and reliable techniques to design, develop, test and verify these systems. With model based design and development becoming a trend, industries use design and simulation tool sets like MATLAB and Mathematica.

It is considered as the de facto standard in avionics and auto motives. It enables modeling, simulating and analyzing dynamic systems. It provides a wide range of library blocks, for example, Math Operation blocks, Logic and Bit Operation blocks, Signal Routing blocks, to name a few. It supports linear and non-linear systems, modeled in continuous time, sampled time or a hybrid of two. Simulink is basically an add-on library to MATLAB with a number of blocks like Integration block, Summation block etc with the help of which one can design and capture the dynamic behavior of a system under consideration.

To capture the discrete control states, one generally uses Stateflow which is a component of Simulink. Stateflow is an interactive graphical design tool that works with Simulink to model and simulate event-driven systems, also called reactive systems. It provides a graphical editor on which the Stateflow graphical objects can be dragged and dropped from the design palette can be put to create finite state machines. It allows hierarchical state machine diagrams, State charts to be combined with flowcharts. SL/SF is a widely accepted tool in the industry for model based development of systems.

This environment supports hierarchical development of complex controller designs and provides a rich set of high level and customizable computational and control blocks suitable for hybrid control systems. A wide variety of application specific block-sets available with SL/SF environment enable easy development of control systems in various domains. SL/SF models help in early design exploration, simulation, automatic code generation for different hardware/software platforms. Simulink organizes its blocks into block libraries according to their behavior. The following are some of the block libraries :-

Sources Library:- This library contains blocks that generate the signals.

Sinks Library:- This library contains blocks that display or write block output.

Linear Library:- This library contains blocks that describe linear functions.

Non Linear Library:- This library contains blocks that describe Non-linear functions.

Connections Library :- This library contain blocks that allow multiplexing and demultiplexing, implement external Input/Output, pass data to other parts of the model, create

subsystems, and perform other functions.

2.4 Stateflow

Stateflow is an interactive graphical design tool that works with Simulink to model and simulate event-driven systems, also called reactive systems. It provides a graphical editor on which the Stateflow graphical objects dragged from the design palette can be put to create finite state machines. We know that Stateflow [7] model is a graphical representation of a finite state machine where States and transitions form the basic building blocks of the system.Stateflow extends the ease of modeling by adding hierarchy, parallelism, events, actions and history. Stateflow blocks are treated as standard Simulink blocks and hence can be integrated in the Simulink models.

States form the basic object in Stateflow and reflect modes in a dynamic system. States can be active or inactive. Active state means that the Stateflow is in that mode. Multiple states can be active during a state, depicting parallelism. Events and conditions cause the states to change from inactive to active states and vice versa. States in Stateflow can have hierarchy. For example, gear state is the parent of first, second, third and fourth state in the Shift Logic block. The events and transitions between them are contained in gear state. Every state has a decomposition that dictates what kind of substates it can contain. All substates must be of the same type as of superstate‘s decomposition. There are 2 types of state decompositions namely OR and AND decomposition. In OR decomposition, only one substate of the parent state can be active at a time. For example, On state has OR decomposition in the Mode example where Init, Active and Inactive are its OR substates. OR substates have solid borders. In AND decomposition, the child states are parallel states. All the substates are active at the same time. States have labels which denote the state's name, entry actions, during actions, exit actions and on event actions.

1. Entry actions define the actions to be taken when the state is entered or activated.

2. During actions define the set of actions to be taken when the state is already active and some event occurs

3. Exit actions define the set of actions to be taken when the state becomes inactive from active.

4. On event actions define the actions to be taken when a state is active and the mentioned event occurs.

Transition in Stateflow means a jump from some source state to some target state. Transitions have a label associated with it. The label can consist of an event, a condition, a condition action and/or a transition action having the following format.

event[condition]condition action / transition action

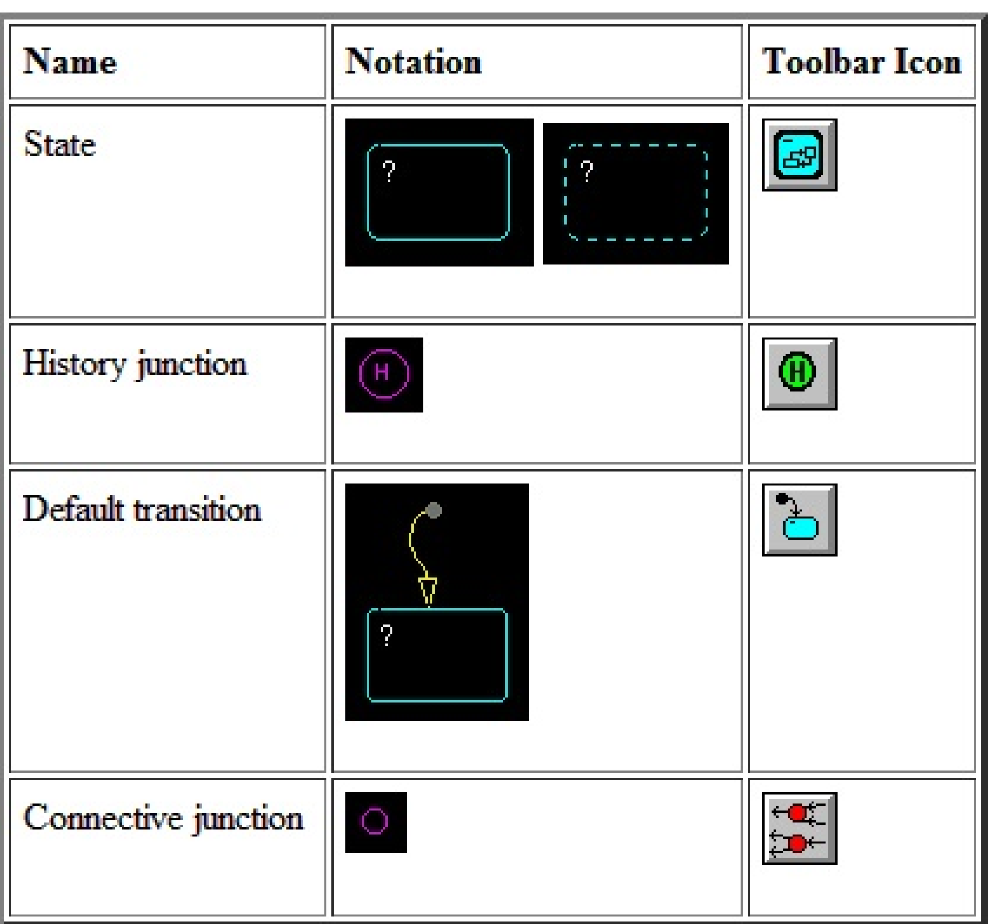
1. Event specifies the event that should cause the transition to occur.
2. Condition specifies a boolean expression that needs to be evaluated to true for the transition to take place.
3. Condition action specifies the action to be immediately executed when the condition evaluates to true.
4. Transition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

There can be different types of transitions in Stateflow like inner transitions, transitions between substates etc. For the current project, we consider only flat Stateflows without junctions, history junctions. Hence we do not discuss about them here.

Stateflow notation consists of a set of graphical objects and a set of non graphical text-based objects and the defined relationships between those objects. Graphical objects include States, Transitions, History junction, Connective Junction. Non-graphical objects include Events and Data elements. Some are shown in Figure 2.3.

2.5 Basic concepts of Slicing

In this section we discuss the background concepts including the basics of slicing. Program slicing is a technique for extracting statements of a program that affect or are affected by the values of a set of variables at a specific point in the program. The extracted statements are called a slice. The specific point in the program and the variables of interest are usually referred to as a *slicing criterion*. Slices are either static or dynamic as well as executable or not executable. An overview of program slicing can be found in [8] [9]. While the original slicing approach [9] was based on solving data flow equations, slicing algorithms nowadays are using program dependence graphs (PDG) [10] [11]. A PDG is a directed, rooted graph with nodes representing the program statements and edges the representing dependencies among them. Using PDGs, slicing can be mapped to a reachability problem. Depending on the direction of edge traversal, a slice is called a *forward slice* or *backward slice*. While



Name	Notation	Toolbar Icon
State		
History junction		
Default transition		
Connective junction		

Figure 2.3: Notations in the Stateflow

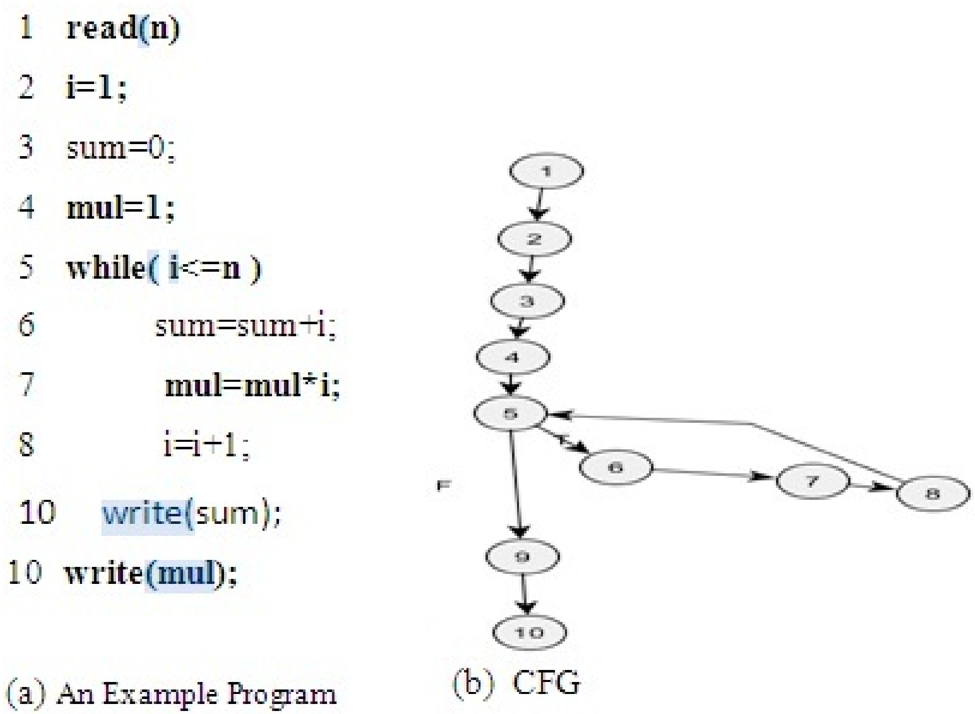


Figure 2.4: An Example Program and its CFG

backward slices extract the statements that influence the program point given in the slicing criterion forward slices contain the statements that are influenced by the slicing criterion in the further execution of the program.

To build the PDG for a program, a dependence analysis has to be carried out first. A PDG consists of two types of dependencies: data dependence and control dependence. These dependencies are defined in terms of the control flow graph of a program. Data dependence is usually given by DEF-REF-relations of variables during the traversal of the control flow graph (CFG) on all paths.

Data Dependence: In a CFG a node j is said to be data dependent on a node i if there exists a variable x such that

(i) x is defined at i

(ii) x is referenced/used at j. And

(iii) There exists a path from i to j where x is not redefined at any node in that path.

Control dependence between two nodes in a CFG is usually given in terms of *post-dominance*. A node i is post dominated by a node j if all paths from i to the exit node pass through j. In Figure 2.4 node 5 post dominates node 4.

Control Dependence: A node j is control dependent on a node i if

(i) There exists a path from i to j such that j post dominates every node in the path excluding i, and

(ii) i is not post-dominated by j

In other words, a node j is control dependent on a node i if i have at least two outgoing edges and j is not in all of the paths to the exit node starting from these edges. In Figure 2.4 the node 7 is control dependent on node 5. Control dependence can be determined by calculating the post-dominator tree for a CFG or for programming languages with a well structured control flow directly from the abstract syntax tree.

In Figure 2.4, we give a small example program that is often used to demonstrate slicing. It calculates the sum as well as the product of the first n positive integers. The highlighted statements are contained in a backward slice for the slicing criterion C(10, mul) where line 10 is the program point and 'mul' is the set if variables of interest.

2.6 Objective

- To propose an algorithm that can generate test sequences for Simulink/Stateflow models.
- To generate test sequences for complex real time embedded control systems.
- Computing slices for Simulink/Stateflow models.

Chapter 3

Review of Related Work

In this section First we give the literature work of test case design and after that we are giving the slicing literature review.

3.1 Literature review on testing of Simulink Models

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. Many approaches are there for this purpose. One of these approach is T-Vec Tester [12] , generates test cases automatically from the domain testing theory. It is based on the premise that if there is no coincidental correctness, then test cases that localize the boundaries of domains with arbitrarily high precision are sufficient to test all the points in the domain. This approach relieves engineers from many manual tasks and reduces the possibility of manual error. From a customer or FAA-certifier(Federal Aviation Administration) perspective, the T-VEC method and automation make the development process very systematic.

Second one is Reactive Systems Reactis Tester [13], by using guided simulations and heuristics without explanation. These two approaches are limited regarding the length of generated input signals, model size and complexity leads to lower structural coverage.

An academic approach is described in the PhD thesis of Zhan [14]. Therein a novel search-based approach to covering particular structural elements of Simulink models is proposed.

1. Initially, probes are instrumented into the model under investigation in order to enable

monitoring of the data flow resulting from the execution with the previously generated inputs.

2. The monitored data flow is used to assess the capability of the applied inputs to cover certain structural elements by means of specific objective functions.

3. With rising model size the complexity of these objective functions also rises, forming complicated search landscapes which are hard to survey.

4. Furthermore, the examination of the entire model for investigating a specific test goal requires execution of the complete model. This may lead to long run times due to the usually big magnitude of real-world models.

5. Generally, sequences of input data (signals) need to be generated in order to be able to deal with internal states. Zhan proposes the objective value of one test case - a signal consisting of several time steps - to be calculated from the objective values of each single time step.

6. This way, the generated signals exhibit a rather random shape, which may become a problem for complex models.

7. This is true especially for models of real-world embedded systems since the inputs to these systems are composed of gradually changing signal patterns in most instances.

Draw back of Zhan's approach: Furthermore, only a small subset of Simulink blocks has been considered and, more profoundly, Stateflow diagrams are disregarded completely.

Andreas Windisch [15] extended the Zhan's work, by considering the task of structural testing of Simulink models containing Stateflow diagrams, a number of difficulties arise that need to be addressed.One of the main problem is the size of the model under test and the resulting complexity. Other major challenges are the signal generation as well as the coverage of Stateflow diagrams.

The general contributions of this research can be abstracted as the improvement of software quality assurance for continuous functional models by automating the test case generation process by means of structural testing. When completed, a structural testing environment is expected to be available that exhibits the following features:

- Automatically generating relevant and realistic test cases for Simulink/Stateflow models fulfilling structural coverage criteria.

- Allowing for effective and efficient test data generation regardless of the model size and complexity.

In this approach one major drawback is it is not applicable for realistic complex models.

Gadkari [16] have translated Simulink/Stateflow to a formal language and generated test cases based on model checking. In this paper, he described our experience in test generation using model checking for the Simulink/Stateflow (SL/SF) models of two automotive controller examples. Model checking based test generation is non-trivial since the SL/SF models have to be first translated into a formal language to serve as an input for the model checker tool.

Moreover, to handle the size and complexities of the industrial designs the translation has to make use of various abstractions yet preserve the semantics of the original model relevant for test generation. He provided an outline of the translation scheme used for translating the SL/SF models into a formal language called SAL. Preliminary results indicate that model checking based test generation, in conjunction with suitable model abstractions, can yield better results in terms of coverage and efficiency of test-cases as compared to the conventional approaches based on simulation and random data generation.

Meng Li and Ratnesh kumar [17] introduced a recursive method to translate a Simulink/Statefl diagram to an Input/Output Extended Finite Automata 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. [18] proposed one approach to test suite design for code generation tools. They describes 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.

Code Generator Testing Approach

1. A formal specification of a code generator transformation is created as a graph rewriting rule.

2. The graph transformation rule is then used as a blueprint to describe the possible input domain of a transformation rule with the Classification-tree Method.

3. In order to stimulate all possible simulation pathways through a given test model, we employ structural coverage metrics on model level. Automated test vector generation with regard to these coverage measures is used to find a selection of input data which achieves full structural model coverage. This can, to a large extent, be automated using tools such as Reactis.

4. After code generation has been carried out, a similar approach is followed on code level: this time structural testing is used to create a second set of test vectors, which guarantees complete structural coverage of the C code generated.

5. After test vector sets for model and code coverage have been generated, both test data sets are merged together. This is necessary because the control flow of the model and the code could be different. On the one hand, optimization techniques could omit or melt branches of the model.

6. Finally, the model and the code outputs are compared. If these are sufficiently similar for one and the same test vector, this is an indication that the code generator and the other tools used (e.g. compiler, linker) are working correctly. If, however, they are (substantially) different, one can conclude that this is due to an incorrect implementation of the code generator, a problem with one of the other tools involved, a faulty test model or an incomplete specification of the optimization (incorrect graph transformation).

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

and Ratnesh kumar approach is based on Finite automata. Stateflow is better approach than the Finite automata. So our approach is good.

3.2 Literature review of Slicing of Simulink Models

In the absence of any directly existing reflected work, we compare our work with some of the existing works which are close to our work. In the past ten years, a number of approaches have been published on slicing various kinds of models. Most of these publications are on slicing state-based models such as extended finite state machines or Statecharts. A comprehensive overview about slicing techniques for state-based models are given in [18]. To the best of our knowledge, no approach for slicing Simulink/Stateflow has been published yet.

In 2000, the tool sliceMDL [19] was developed by Krogh. Since this was a project for an industrial sponsor who wanted to retain the rights to this work, we have not been able to evaluate our approach against this tool. There are also no publications on this work, but only a few documentation on the web page. According to this, sliceMDL uses the signal flow between blocks to slice a model but does not consider execution context propagation and control dependence.

Argos and UML Statecharts are graphical notations that allow the modeling of hierarchical state machines. Ganapathy et al. [20] introduced slicing for Argos. Wang et al. [21] presented an approach for slicing UML Statecharts based on a transformation to extend the hierarchical automata which later was improved by van Langenhove [22]. Both approaches do not use dependence graphs.

Korel [23] and Androutsopoulos [24] [25] [26] developed slicing approaches for extended finite state machines (EFSMs). Both of them adapted the control dependence as defined in imperative programming languages to EFSMs. While [23] requires an exit state, [24] [25] used control dependence which is nontermination insensitive. For slicing, they used dependence graphs. In [26], another approach for slicing of EFSMs was presented where the EFSMs were reduced according to input variables from the environment. In [25], slice sizes for EFSMs were compared for different definitions of control dependence. They, also compared the average slice size of EFSMs to an average slice size for imperative programming languages [27] and reported that that the average slice size for EFSMs was larger than that of imperative programming languages.

Reicherdt and Glsner [28] proposed an approach for slicing of MATLAB Simulink Mod-

els. Their approach was based on dependency graphs. They had given slicing criteria based on Conditional Execution Contexts. But they had not considered the Stateflow part of the model. In our approach we have considered both Simulink and Stateflow models. So the slices computed by our approach are more correct and precise.

Chapter 4

Generating Test sequences for Simulink/Stateflow models

In this chapter, we will first of all explain the implementation details, and talk about the tools we have used for the implementation. And finally, we will discuss our approach for generating test sequences for Simulink/Stateflow models.

4.1 Generating Test sequences for Simulink/Stateflow models

Our method of approach is Generating Test sequences from SL/SF models, First we are generating graphs from the Graph Generator. From that graphs using Test Sequence Selector we are selecting test sequences. The approach is shown in Figure 4.1.

Our method of generating test sequences is based on using dependency 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.In Stateflow the dependency exist is control dependency.

Output dependency graph: 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.

Control Dependency graph: 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.

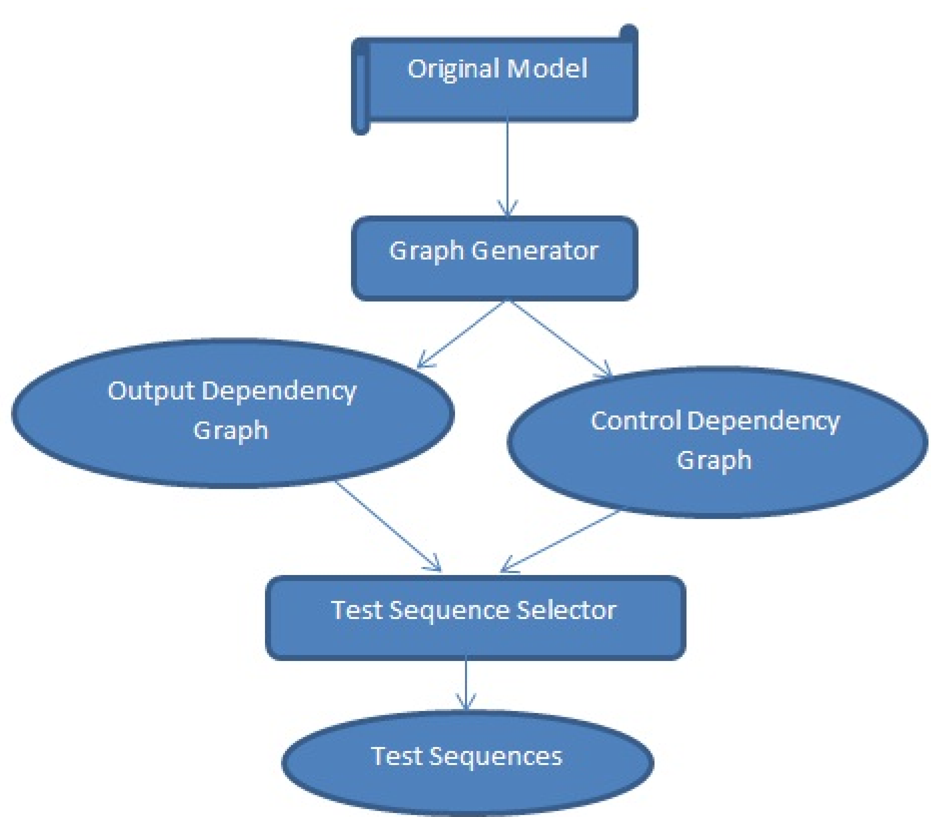


Figure 4.1: Block Diagram of our Approach

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 algorithm 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, Stateflows are represented using separate Graphs. The number of graphs are constructed based on the hierarchy of subsystems and Stateflows in a SL/SF model.

Algorithm Details: This graph takes 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.

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

Stateflow_graph_construction(file path);

End

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 consider the adjacent blocks and note down in the adjacency matrix. Using the adjacency matrix draw 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.

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

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 draw the graph in the dotty file using Graphviz tool. After generating the graph we use adjacency matrix to generate the test sequences. These test sequences are used to generate the test cases. If we want to test efficiently we have to create the test cases for each and every test

sequence.

4.2 Implementation Details

1. Language used for implementation is Java.
2. Inputs to the tool- SL/SF models: Original Matlab model file having .mdl extension.
3. Outputs of the tool a. Intermediate graph model representations b. Test sequences

4.3 Tools used for the implementation

We have used the following tools for implementation.

1. NetBeans
2. Graphviz
3. Matlab Simulink
4. Matlab Real Time Workshop

Ntebeans

NetBeans is an integrated development environment (IDE) for developing primarily with Java, but also with other languages, in particular PHP, C/C++, and HTML5. It is also an application platform framework for Java desktop applications and others. The NetBeans IDE is written in Java and can run on Windows, OS X, Linux, Solaris and other platforms supporting a compatible JVM. The NetBeans Platform allows applications to be developed from a set of modular software components called modules. Applications based on the NetBeans Platform (including the NetBeans IDE itself) can be extended by third party developers. This is an open source software, we can download it from the site "https://netbeans.org/downloads/", and we can directly run it. For How to use Netbeans we can use its manual that is provided with Netbeans.

Framework for simplifying the development of Java Swing desktop applications. The NetBeans IDE bundle for Java SE contains what is needed to start developing NetBeans plugins and NetBeans Platform based applications; no additional SDK is required.

Applications can install modules dynamically. Any application can include the Update Center module to allow users of the application to download digitally signed upgrades and

new features directly into the running application. Reinstalling an upgrade or a new release does not force users to download the entire application again.

The platform offers reusable services common to desktop applications, allowing developers to focus on the logic specific to their application. Among the features of the platform are:

- User interface management (e.g. menus and toolbars)
- User settings management
- Storage management (saving and loading any kind of data)
- Window management
- Wizard framework (supports step-by-step dialogs)
- NetBeans Visual Library
- Integrated development tools

NetBeans IDE is a free, open-source, cross-platform IDE with built-in-support for Java Programming Language.

Graphviz

Graphviz is an open graph visualization tool. We have used this tool in our project to visualize our intermediate outputs in a better way i.e. in the form of a pictorial graph instead of adjacency matrices or adjacency lists.

Matlab Simulink

MATLAB Simulink/Stateflow (SL/SF) is a high level model designing tool very popular in many industrial application domains. It enables modeling, simulating and analyzing dynamic systems. It provides a wide range of library blocks, for example, Math Operation blocks, Logic and Bit Operation blocks, Signal Routing blocks, to name a few. It supports linear and non-linear systems, modeled in continuous time, sampled time or a hybrid of two. Systems can also be multirate, i.e, have different subsystems that are sampled or updated at different rates. Simulink is basically an add-on library to MATLAB with a number of blocks like Integration block, Summation block etc with the help of which one can design and capture the dynamic behavior of a system under consideration.

Systems can be modeled in Simulink by creating a network of blocks dragged from Simulink block library and dropped into the GUI editor and connecting the appropriate ports. We have used this tool to develop dummy SL/SF models for testing our tool.

Matlab Real Time Workshop The Real-Time Workshop, for use with MATLAB and Simulink, produces code directly from Simulink models and automatically builds programs that can be run in a variety of environments, including real-time systems and stand-alone simulations.

With the Real-Time Workshop, you can run your Simulink model in real-time on a remote processor. The Real-Time Workshop also enables you to run high-speed stand-alone simulations on your host machine or on an external computer. Using the rapid prototyping process, you can shorten development cycles and reduce costs. You can also use the Real-Time Workshop to implement hardware-in-the-loop (HIL) simulations. The Real-Time Workshop provides a real-time development environment that features:

- A rapid and direct path from system design to implementation.
- Seamless integration with MATLAB and Simulink.
- A simple, easy to use interface.
- An open and extensible architecture.
- A fully configurable code generator? virtually every aspect of the generated code can be configured by using the Target Language Compiler?.
- Fast design iterations by editing block diagrams and automatically building a new executable.

The package includes application modules that allow you to build complete programs targeting a wide variety of environments. Program building is fully automated. The generated code (i.e., the model code) is by default highly optimized and fully commented C code that can be generated from any Simulink model, including linear, nonlinear, continuous, discrete, or hybrid models. All Simulink blocks are automatically converted to code, with the exception of MATLAB function blocks and S-function blocks that invoke M-files. You must rewrite these blocks as C MEX S-functions if you want to use them with the Real-Time Workshop.

4.4 Implementation Results

In this section, we explain the working of our algorithms by taking the Fan example using a Simulink/Stateflow model, as described below and it is shown in Figure 4.2.

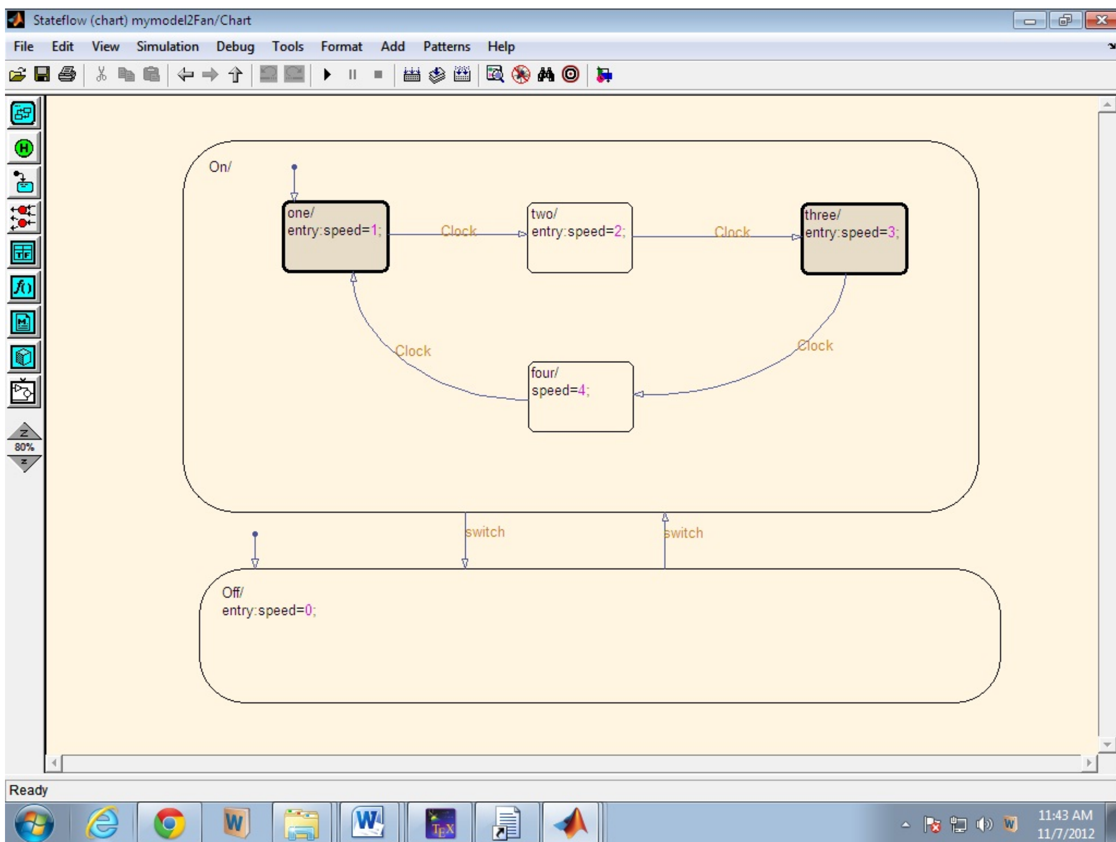


Figure 4.2: Stateflow of fan model

4.4.1 An illustartive Example

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. In this time the speed of the fan is ‘0’. When the switch signal become ‘on’ the transition goes to work and control goes to On/ state. On/ state having hierarchy, it is having substates one/, two/, three/, four/. In this 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 4.2. When drawing the graph we are representing the default transition by using the START node.

The Figure 4.2 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 4.3. In the On/ node it

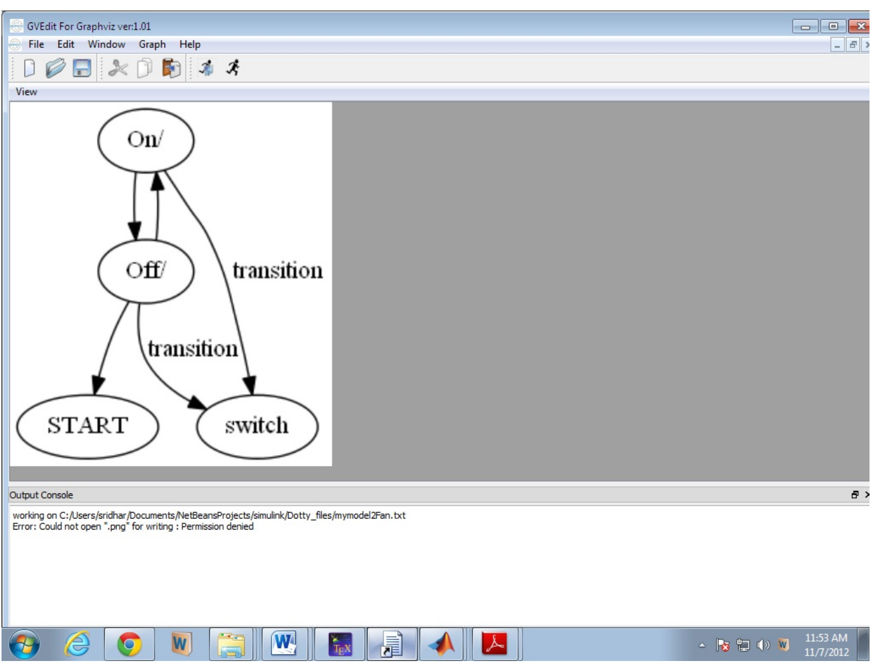


Figure 4.3: Top level graph

is having sub nodes for the representation of sub nodes. Our algorithm generates one more graph which is shown in Figure 4.4.

From this graph we will generate test sequences based on the state coverage, transition coverage etc.

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

Top level test sequences:-

1. Start $\rightarrow off/(ifthesignalisnotgenerated)$
2. Start $off/ \rightarrow on/ \rightarrow off/(ifthesignal$ is generated)

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

1. start $\rightarrow one/ \rightarrow two/ \rightarrow three/ \rightarrow four/ \rightarrow one/(Inthiscaseitiscovering$ all states)

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

Top level test sequences:

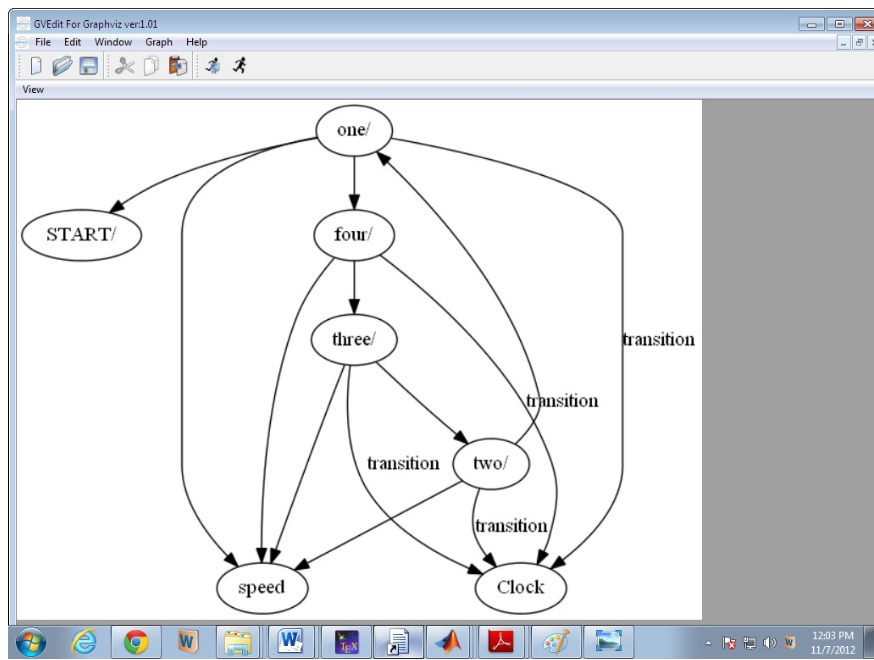


Figure 4.4: Secondary level graph

1. $Start \rightarrow off/(ifthesignalisnotgenerated)$

2. $Start \rightarrow off/ \rightarrow switch \rightarrow on/ \rightarrow switch \rightarrow off/(ifthesignalisgenerated)$

Secondary level test sequences:

1. $Start \rightarrow one/ \rightarrow clock \rightarrow two/ \rightarrow clock \rightarrow three/ \rightarrow clock \rightarrow four/ \rightarrow clock \rightarrow clock(inthiscaseitcoveringallstatesandalltranstions)$

The transition coverage is the better than the state coverage it covers all the states and extra it covers all transitions. I generated test sequences from Simulink model that is shown in Figure 4.5. When we are sending this model as input to the our algorithm, it is giving graph as output, the graph is shown in Figure 4.6.

From Figure 4.6 we generated the test sequences are:

1. $Start \rightarrow tg \rightarrow Abs \rightarrow Gain \rightarrow sum2 \rightarrow switch \rightarrow outputport$

2. $Start \rightarrow u \rightarrow switch \rightarrow outputport$

3. $Start \rightarrow u \rightarrow sum2 \rightarrow switch \rightarrow outputport$

4. Start $\rightarrow$ u $\rightarrow$ $product1 \rightarrow sum1 \rightarrow$ switch $\rightarrow$ $outputport$

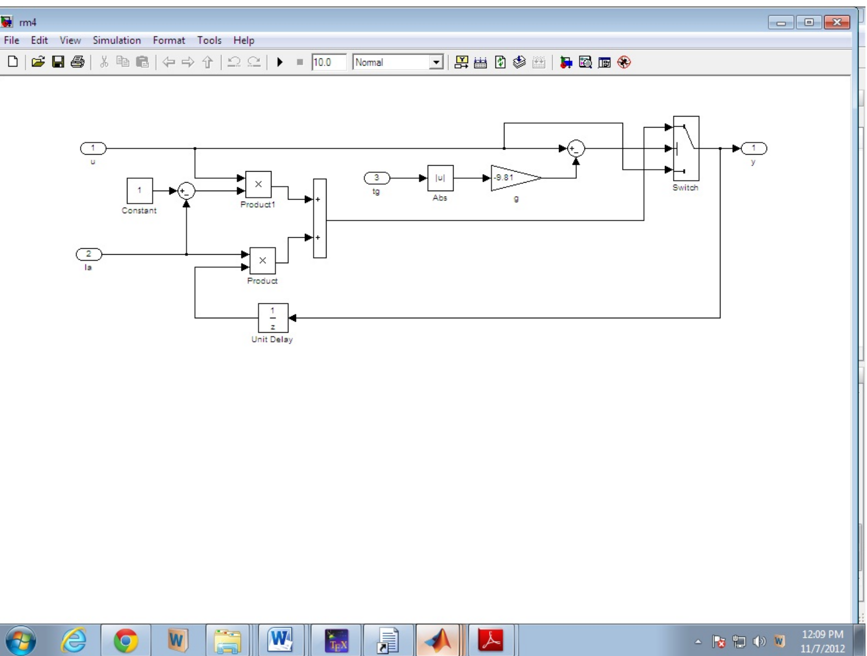


Figure 4.5: Example of Simulink model

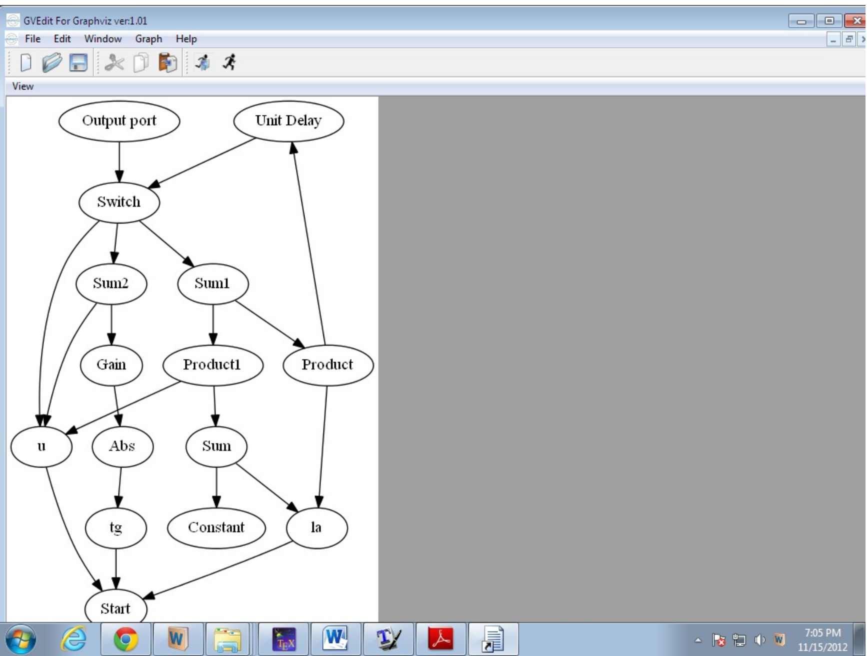


Figure 4.6: Graph of Simulink model

5. $Start \rightarrow la \rightarrow sum \rightarrow product1 \rightarrow sum1 \rightarrow switch \rightarrow outputport$

6. $Start \rightarrow u \rightarrow switch \rightarrow unitDelay \rightarrow product \rightarrow sum1 \rightarrow switch \rightarrow outputport$

7. $Start \rightarrow u \rightarrow sum2 \rightarrow switch \rightarrow unitdelay \rightarrow product \rightarrow sum1 \rightarrow switch \rightarrow outputport$

8. $Start \rightarrow tg \rightarrow Abs \rightarrow Gain \rightarrow sum2 \rightarrow switch \rightarrow unitdelay \rightarrow product \rightarrow sum1 \rightarrow switch \rightarrow outputport$

9. $Start \rightarrow u \rightarrow product1 \rightarrow sum1 \rightarrow switch \rightarrow unitdelay \rightarrow product \rightarrow sum1 \rightarrow switch \rightarrow outputport$

10. $Start \rightarrow la \rightarrow sum \rightarrow product1 \rightarrow sum1 \rightarrow switch \rightarrow unitdelay \rightarrow product \rightarrow sum1 \rightarrow switch \rightarrow outputport$

11. $Start \rightarrow la \rightarrow product \rightarrow sum1 \rightarrow switch \rightarrow unitdelay \rightarrow product \rightarrow sum1 \rightarrow switch \rightarrow outputport$

4.4.2 Complexity Analysis

The complexity of the model under [1] 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 graph 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 in 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 for the above such models will be O(n^2). 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 subsytems 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 toplevel 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 adjacency

matrix. The size of 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 graph. Let the number of states be 'S' and transitions be 'T'. Then 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. Complexity is O(1+m(k+1))B^2+n(S+T). space complexity for every level is O(B^2).

Chapter 5

Slicing of Simulnik/Stateflow models

In this section we are giving the details about the dependencies in SL/SF models. Steps to find the slices of Simulink models.

5.1 Dependencies in SL/SF models

Simulink/Stateflow is a data flow oriented graphical notation where data flow is by the structure and control flow has to be calculated. In this section, we present how to determine data and control dependencies for Simulink/Stateflow models.

Data Dependence in Simulink: Data flow in Simulink is given by the signal lines. So data dependence can easily be derived from an observing the signal lines.

A block b2 is data dependent on a block b1 if i) b1 and b2 are connected by a signal line L and

ii) L starts with an output of b1 and ends in an input of b2.

Control Flow in Simulink: Basically the control flow in Simulink is modeled using MathWorks [29]

a) Conditional subsystems- Conditional subsystems are Enabled, Triggered, Enabled , Triggered, Action and Function-Call subsystems.

b) Loop subsystems- Loops like while and for in Simulink are realized by atomic subsystems.

c) Multi Port Switch and Switch blocks- Switch or MultiPortSwitch blocks can also be used to model the control flow in Simulink. To optimize the simulation of a model, and avoid the unnecessary execution of model parts. The Simulink environment offers Conditional Execution Behavior. Control Dependence and Conditional Execution Contexts: For each

atomic subsystem a new execution context is created. Since the all the conditional and loop subsystems being atomic, an execution context (EC) is created for each of these and the blocks contained in each of these subsystems are assigned to their ECs and are scheduled within. Conditional subsystems and loops are executed only if the condition given by the special port or iterator block evaluates to true. Hence the corresponding ECs is called Conditional Execution Contexts (CEC). In the following, we refer to this special port or iterator block as predicate block.

Control Dependence in Simulink: In a model m with the root context e^m containing the blocks b1 and b2, block b2 is control dependent on block b1 if

1) b2 is within a conditional EC e $\neq$ em and

2) b1 is the predicate block controlling the execution of e.

The blocks contained in em are control dependent on the simulation loop. Data Dependence in Stateflow: Data dependence edges are used to represent the dependency of a state on the data items associated with a Stateflow model. This includes all the data variables used in the entry, during, and exit sections of any state. The data variables can also associate with conditions, condition actions, or transition actions in any state transition.

Control dependence in Stateflow: Control dependence in Stateflow depends on the transitions between the states. The first Stateflow execution starts from the default transition. A control dependence edge represents an inter stage dependency due to the change of state in the Stateflow model. A control dependence may arise in the following three ways:

Case1: Control dependence arises when there is no transition label associated with the transition between two states.

Case2: When an outgoing transition from a state has a transition label, then control dependency arises between the state and the predicate node.

Case3: Control dependency arises when a state has an ?exit action? and its outgoing transition is associated with condition action, and transition action. This is because once the condition of a transition is satisfied, then the condition action takes place. Before executing the transition action and jumping to the next state, the exit action of the current state takes place after the transition execution. This results in control dependence arising between the predicate and the state node with label having the name of the predicate node and state.

5.2 Slicing of Simulink/Stateflow models

In this section, we present our slicing approach for Simulink/Stateflow models using dependence graphs. First, we define the slicing criterion and the slice of a Simulink/Stateflow model. Then we show, how we calculate dependencies which includes the calculation of the conditional execution contexts. Finally we show how we build up the dependence graph and compute static slices for the model.

Simulink/Stateflow slicing criterion A slicing criterion for Simulink/Stateflow model may be any block except a subsystem or any state in the Stateflow or any variable within the Stateflow.

Simulink/Stateflow Slice: A slice of a Simulink/Stateflow model m w.r.t. Slicing criterion C is a model m1 that

1) Contains only those blocks and states from the Stateflow that are relevant to the slicing criterion C (forward slice or backward slice) or

2) contains only those blocks and states from Stateflow to which the slicing criterion C is relevant and that

3) Preserves the hierarchical structure of the Simulink/Stateflow model.

5.2.1 Computing conditional execution contexts

Richerdt et al. [28] has presented an approach for computing the Conditional Execution contexts for Simulink models. In their algorithm, they have considered different Conditional Execution Contexts for every subsystem. If the subsystem is itself a Stateflow model then also they have considered it as virtual block. We are adding the Stateflow block to the execution context and we create one separate execution context.

Steps for constructing CEC of Stateflow model:

Step 1. create the separate Execution Contexts for every chart in Simulink.

Step 2. Within the chart, locate all the nodes and identify the default node.

Step 3. Within the chart, locate the transitions and add them to the Execution Contexts.

5.2.2 Building the dependence graph

After all the blocks have been added to an execution context, we build the dependence graph. First, we create nodes for each block that is not a subsystem. Then, we add the data dependence edges between the nodes. Figure 5.3 Shows the dependence graph for our example Simulink/Stateflow model given in Figure 5.1 and Figure 5.2. In Figure 5.3 this

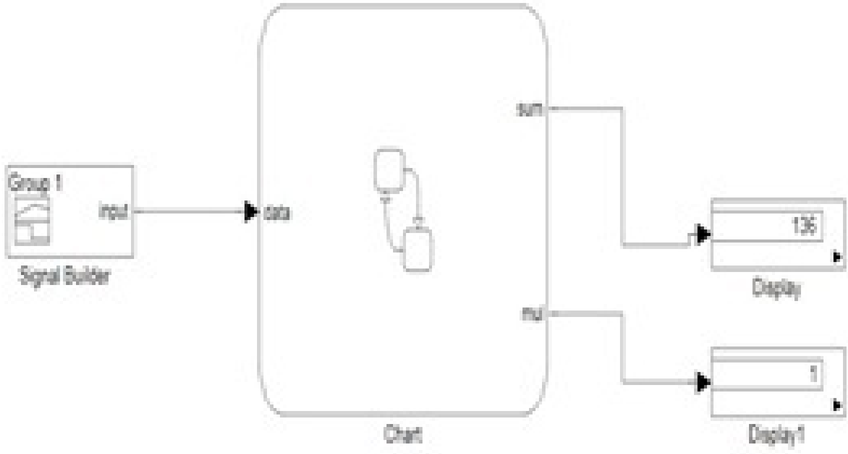


Figure 5.1: An Example Simulink/Stateflow model

figure thick lines represent data dependence in Simulink, dotted bubble lines represent data dependence in Stateflow, dotted arrow represent control flow in Stateflow.

5.2.3 Computing the Simulink/Stateflow Slice

We compute the slice for a model by performing a reachability analysis in the forward or backward direction and marking relevant nodes. Then we remove the unmarked blocks and finally all empty subsystem of the model.

5.3 Slicing Example

Figure 5.1 shows a Simulink/Stateflow model. It calculates the 'sum' when the input values are positive and displays the result as the output. If the input value is negative it calculates the multiplication of those numbers and display it as the output. This is modeled using a chart (which is Stateflow). A chart subsystem is executed as long as the signal is supplied to the chart. To execute a chart we have to give input through Signal Builder. Only subsystem blocks can contain other blocks. The ports of a subsystem block correspond to Port blocks within the subsystem. A number is assigned to each Inport or Outport block which corresponds to the vertical position of the input of the subsystem block. In our example chart is a subsystem. Within the subsystem, the Stateflow model is present. We have shown the Stateflow part of the above example model in Figure 5.2.

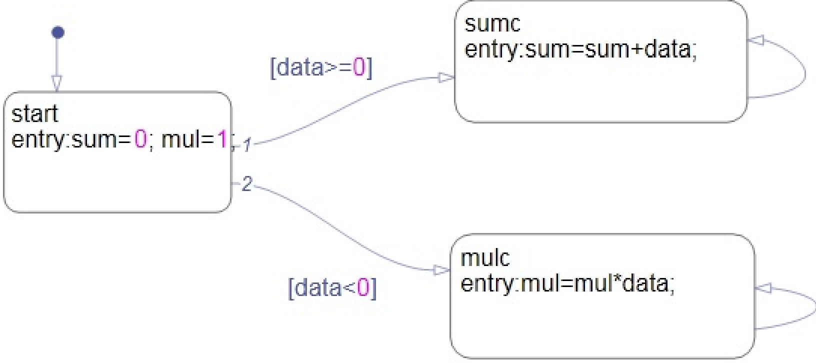


Figure 5.2: Stateflow part of the Model

In this section, to implement our approach we are considering the example model which is shown in Fig 5.1. We have taken this model as input and computed Conditional Execution Contexts. By using the CEC we are constructing the model dependence graph for the SL/SF model given in Figure 5.1 and Figure 5.2 is shown is Figure 5.3.

After constructing the graph, we have taken the slicing criteria as 'mul' and performed a backward traversal of the dependence graph. The nodes reached during the traversal are shown as marked nodes on Figure 5.4. These marked nodes constitute the static slice w.r.t to the slicing criterion 'mul'.

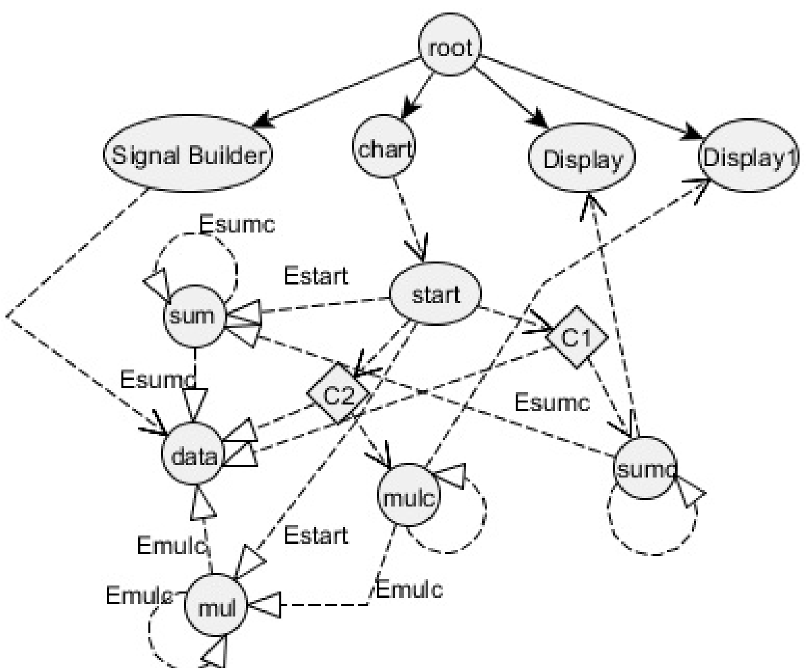


Figure 5.3: The model dependence graph for the SL/SF model in Figure 5.1 and Figure 5.2

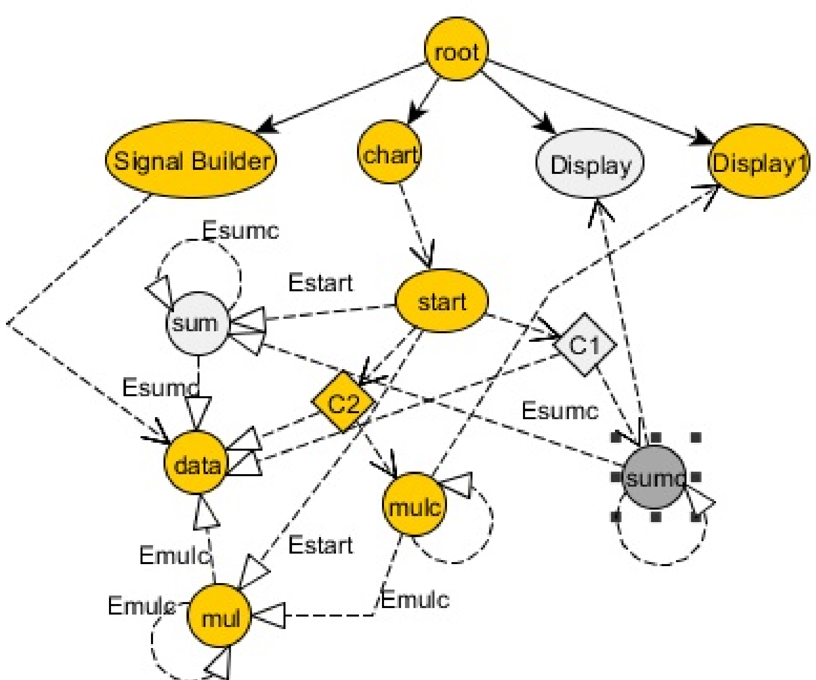


Figure 5.4: Marked model dependence graph showing the static slice w.r.t. the slicing criterion 'mul'

Chapter 6

Conclusions and Future Work

We proposed a methodology to generate test sequences from SL/SF models. First we have constructed the model in the MATLAB environment by using Simulink/Stateflow designing tool. For that model we developed a dependency graphs, and from that graphs we developed a test sequences. We have presented a slicing approach to slice the Simulink/Stateflow models, With slicing the complexity of a model can be reduced to a given point of interest by removing unrelated model elements, thus paving the way for subsequent static quality assurance methods.

However, the implementation of our work relies on some basic assumptions that we have made. They are the following:

1. SL/SF models contain only states and transitions. They do not contain other advanced features of Simulink/Stateflow models like the history blocks, connective junctions, graphical functions, S-functions, truth tables, etc which may be present within the Stateflow chart.

2. The concept of parallel states are not considered by our tool.

As part of the future work, we needs to handle parallel states. In addition to this, other features of Simulink/Stateflow models like history blocks, connective junctions, truth tables, graphical functions, etc should also be handled.

We can use slicing for variety of applications.We briefly outline a few of these in the following.

Change impact visualization When a change is made to a state/block/transition in an SL/SF model, determination of the other blocks/states/transitions that are affected due to this change is an important problem that every designer and maintainer faces.If we

change a block X in SL/SF model and if we want to know all other blocks that are affected we proceed as follows: take the block X as slicing criterion, and perform forward slice. All the nodes that are present in the slice are mapped on to their corresponding blocks in SL/SF model.

Identification of block criticality A critical block is a block that can impact large number of other blocks when a change is made to it. The criticality of block can be determined by constructing forward slice based each block of SL/SF model as slicing criterion. Critical blocks to be included in many test cases can result in high system reliability.

Dissemination of Work

1. Adepu Sridhar,Srinivasulu D, and Durga Prasad Mohapatra.Model-Based Test-Case Generation for Simulink/Stateflow using Dependency Graph Approach, In proceedings of *2013 3rd IEEE International Advance Computing Conference (IACC) , Ajay kumar Garg College of Engineering Gaziyabad(U.P)*, Pages: 1415-1420, 22nd- 23rd Feb 2013. IEEE Catalog Number: CFP1339F-CDR, ISBN: 978-1-4673-4528-6

2. Adepu Sridhar,Srinivasulu D Slicing MATLAB Simulink/Stateflow Models, In proceedings of *2013 Springer International Conference on Advanced Computing, Networking, and Informatics (ICACNI),*, 12th -14th June 2013. Accepted for publication

3. Adepu Sridhar, Srinivasulu D and Durga Prasad Mohapatra .Prioritization of test sequences of Simulink/Stateflow models,International journal of Systems, Algorithms, Applications, Special issue on*International Conference on Recent Advances in Sciences and Engineering*, Volume 3, ICRASE13, May 2013, ISSN Online: 2277-2677.

Bibliography

[1] N. Chauhan, *Software Testing Principles and Practices.* 9780198061847, Naresh Chauhan, Assistant Professor, Dept. of Computer Engineering YMCA University of Science and Technology Faridabad: Oxford University Press ,YMCA Library Building, Jai Singh Road, New Delhi 110001, 1st ed., 31st Jan 2010.

[2] R. Mall, *Fundamentals of Software Engineering.* Prentice Hall, 3rd ed., 2009.

[3] C. S. Pasareanu, J. Schumann, P. Mehlitz, M. Lowry, G. Karsai, H. Nine, and S. Neema, “Model based analysis and test generation for flight software,” in *Space Mission Challenges for Information Technology, 2009. SMC-IT 2009. Third IEEE International Conference on*, pp. 83–90, IEEE, 2009.

[4] B. Schätz, A. Pretschner, F. Huber, and J. Philipps, “Model-based development of embedded systems,” in *Advances in Object-Oriented Information Systems*, pp. 298–311, Springer, 2002.

[5] B. Schätz, A. Pretschner, F. Huber, and J. Philipps, “Model-based development,” *Institute Of Informatics, Technical University Munich*, 2002.

[6] “Simulink, stateflow and real-time workshop.” Website, 2003. `http://www.mathworks.com/products`.

[7] Website. `http://dali.feld.cvut.cz/ucebna/matlab/toolbox/stateflow`.

[8] F. Tip, “A survey of program slicing techniques,” *Journal of Programming Languages*, vol. 3, p. 121189, 1995.

[9] M. Weiser, “Program slicing,” (Piscataway, NJ, USA), pp. 439–449, in Proceedings of the 5th International Conference on Software Engineering, IEEE Press, 1981.

[10] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," pp. 177–184, in Proceedings of the first ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 1984.

[11] T. R. Horwitz, Susan and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, pp. 26–60, 1990.

[12] "T-vec." Website. `http://www.t-vec.com/`.

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

[14] Y.Zhan, "A search-based framework for automatic test-set generation for matlab/simulink models," *Software Eng. SE-10*, vol. PhD thesis, December 2005. University of York.

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

[16] A. A. Gadkari, S. Mohalik, K. Shashidhar, A. Yeolekar, J. Suresh, and S. Ramesh, "Automatic generation of test-cases using model checking for sl/sf models," in *4 th MoDeVVa workshop Model-Driven Engineering, Verification and Validation*, p. 33, 2007.

[17] M. Li and R. Kumar, "Model-based automatic test generation for simulink/stateflow using extended finite automaton," in *Automation Science and Engineering (CASE), 2012 IEEE International Conference on*, pp. 857–862, IEEE, 2012.

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

[19] B. H. Krogh, "slicemdl." Website. `http://www.ece.cmu.edu/_webk/TVG/downloads.html`.

[20] V. Ganapathy and S. Ramesh, "Slicing synchronous reactive programs," *Electronic Notes in Theoretical Computer Science*, vol. 65, No. 5, pp. 50–64, Apr. 2002. Synchronous Languages, Applications, and Programming.

[21] D. W. Ji, Wang and Q. Zhi-Chang, "Slicing hierarchical automata for model checking uml statecharts," (London, UK), pp. 435–446, in Proceedings of the 4th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering, 2002.

[22] S. Van Langenhove and A. Hoogewijs, "System verification through logic tool support for verifying sliced hierarchical statecharts," pp. 142–155, 2007. Springer Berlin Heidelberg.

[23] B. Korel, I. Singh, L. Tahat, and B. Vaysburg, "Slicing of state-based models," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pp. 34–43, IEEE, 2003.

[24] K. Androutsopoulos, D. Clark, M. Harman, Z. Li, and L. Tratt, "Control dependence for extended finite state machines," in *Fundamental Approaches to Software Engineering*, pp. 216–230, Springer, 2009.

[25] K. Androutsopoulos, N. Gold, M. Harman, Z. Li, and L. Tratt, "A theoretical and empirical study of efsm dependence," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pp. 287–296, IEEE, 2009.

[26] K. Androutsopoulos, D. Binkley, D. Clark, N. Gold, M. Harman, K. Lano, and Z. Li, "Model projection: Simplifying models in response to restricting the environment," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 291–300, ACM, 2011.

[27] G. N. Binkley, David and M. Harman, "An empirical study of static program slice size," *ACM Transactions on Software Engineering and Methodology*, vol. 16, Apr. 2007.

[28] R. Reicherdt and S. Glesner, "Slicing matlab simulink models," pp. 551–561, 2012.

[29] MathWorks, "Using simulink, the mathworks inc.." Website. `http://www.mathworks.com/help/pdfdoc/simulink/sl_using.pdf`.