International Journal of Advanced Computer Research (ISSN (print): 2249-7277 ISSN (online): 2277-7970)
Volume-3 Number-4 Issue-13 December-2013

Generation of Branch Coverage Test Data for Simulink/Stateflow Models Using Crest Tool

Sangharatna Godboley[1], Adepu Sridhar[2],Bhupendra kharpuse[3] , Durga Prasad Mohapatra[4], Banshidhar Majhi[5]

Abstract

Automated test suite generation is an optimization technique to reduce test effort and duration. Software Testing has traditionally been one of the main techniques contributing to high software quality and dependability. Testing performance consumes about 50% of software development resources, so any methods aiming at reducing software- testing costs are likely to reduce software development costs. So an automated generation of test cases is highly required. Modelling technology has been introduced into the software testing field. Even though how to perform the testing is difficult task, in this approach we are testing using simulation modelling and by using testing methods we are generating MC/DC test cases. In our approach, the first system is modelled in MATLAB using Simulink/Stateflow tool. After that we are generating code from the Simulink/Stateflow models. We use that code to generate the test data using Concolic tester CREST Tool. Concolic testing is used in avionics systems in an efficient way.

Keywords

Simulation; Stateflow; Branch coverage; CREST Tool; Simulink/Stateflow; Concolic testing e Slicer, Crest Tool, Concolic Testing.

1. Introduction

Embedded Control Systems are now integral parts of many application systems in the areas of Aerospace, Communication, Automobiles, Commercial (computer peripherals, appliances, etc.), and Industrial (machinery) etc.

Sangharatna Godboley, Department of Computer Engineering, ARMIET Shahpur, Thane, India.
Adepu Sridhar, Vignyan University, AP, India.
Bhupendra Kharpuse, Symantec India Pvt ltd Pune, India.
Durga Prasad Mohapatra, Department Computer Science and Engineering, National Institute of Technology Rourkela, India.
Banshidhar Majhi, Department Computer Science and Engineering, National Institute of Technology Rourkela.

The As a result, everyone looking for easy and reliable techniques to design, develop, test and verify these systems. With a model based design and development becoming a trend, industries use design and simulation tool sets like MATLAB and Mathematical. MATLAB Simulink / Stateflow are a high level model designing tool very popular in many industrial application domains. It enables modelling, 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. Simulink is basically an add-on library for MATLAB with a number of blocks like Integration block, Summation block etc. [9]. To capture the discrete control states, generally uses Stateflow which is a component of Simulink. Stateflow is used together with Simulink and optionally with the Real-Time Workshop (RTW). The control behavior that Stateflow models complements the algorithmic behavior modeled in Simulink. Simulink support development of continuous-time and discrete-time dynamic systems in a graphical block diagram environment. Stateflow diagrams incorporate into Simulink models to enhance the new event-driven capabilities in Simulink.

The model must be tested in order to detect faults in the embedded system as early as possible. Exhaustive test is not possible for any system. The test data generation process is very costly and time consuming and error prone when done manually, the automation of this process is highly required [1].

Embedded controller software is usually based on various states and for that states representation often uses Stateflow [10] diagrams utilization of the internal structure of the diagram to generate test cases is important. In our approach we are performing Concolic testing to generate Branch Coverage [13] test data using CREST Tool.

Concolic testing [15] have been introduced as a divergent of symbolic execution where symbolic execution [14] is executed at the same time with concrete testing. The program under test is specially

performed simultaneously for symbolic and concrete values. The constraints are used to incrementally produce test inputs for best path coverage by merging symbolic constraints for a prefix of the path with the negation of a conditional taken by the execution. The advantage of Concolic testing over symbolic testing is the presence of concrete (data and address) values, which is used to reason precisely about complex data structure as well as to simplify the constraints when they go beyond the scope of the underlying constraints solver. A software program is tested by executing test inputs values, generating output values, and checking outputs against the test oracles for correctness. The combination of test inputs and a test oracle forms a complete test case. It is very difficult to produce test Oracle from source code alone.

CREST [8][18][19] Tool is an open source Concolic testing tool for C programs. Empirical study of CREST shows its effectiveness in unit testing programs with 100 to 2000 lines of code. CREST is based on Linux platform and Yices Constraints Solver. It works by feeding the instrumentation code using CIL into a program under test to perform symbolic execution simultaneously with the concrete execution. The generated symbolic constraints are solved using Yices to generate input that drive the test execution down new, unexplored program paths.

2. Basic Concepts

Simulink is an environment for multi domain simulation and Model-Based Design for dynamic and embedded systems. It is an interactive graphical environment and a customizable set of block libraries. Simulink is used for modelling both continuous and discrete Systems. Simulink blocks are divided into different types according to their behaviour.: a) The Sources library: Contains blocks that generate signals, b) The Sinks library: contains blocks that display or write block output, c) The Linear library: Contains blocks that describe linear functions, d) The Nonlinear library: Contains blocks that describe Non-linear functions, e) The Connections library: Contains blocks that allow to connect different parts of the model. This Simulink information is taken from maths works in company website [9]. Stateflow diagram is a graphical representation of a finite state event driven machine. Stateflow is a powerful graphical design and development tool for complex control and supervisory logic problems. Stateflow allows us to use flow diagram notation and state transition notation seamlessly in the same Stateflow diagram. Stateflow uses different kinds of notation established by Harel [2]. Stateflow allows two basic building blocks states and transitions to represent the finite state machine. Stateflow allows hierarchy, parallelism, and history. Because of these characteristics Stateflow is usefulness compare to STDs and bubble diagrams provide. We design the Stateflow machine using Stateflow blocks. If we go for simulation it executes both Simulink and Stateflow portions. Simulink model consists of Simulink blocks, Subsystems, toolbox and Stateflow Block. In the Simulink model Stateflow represented as Chart. A collection of all these charts is called as Stateflow Machine. Chart consists of state, transition, data, events are there. Sample Simulink Example: A sample of the Simulink model which containing a Stateflow diagram in it. When you simulate this model, the generation of the input event from Simulink, Switch, will toggle the activity of the states between Powers on and Power off. In this model Simulink used as an interface for the Stateflow model. In the sample Simulink example Fig. 1, the Stateflow part of the model is shown in Fig.2. The Concolic testing concept combines a concrete constraints execution and symbolic constraints execution to automatically generate test cases for full path coverage. Concolic testing produces test cases by performing the program under test with random values. At the time of execution both symbolic and concrete values are saved for path execution. The next repetition of execution of process forces to select the different path. The Concolic tester chooses a value from the path constraint solver (Yices for CREST Tool) and negates the values to find a new path value. The constraints are inputs for all next execution. This process executes iteratively until exceeds the sufficient code coverage obtained or threshold value.

Let us take an example, Fig.3:
Calculate Density of mass when mass and volume are given. Concolic Tester starts by executing the method with random strategy. Assume that Concolic tester has set the values of mass 91 and volume= -9. During execution time both concrete and symbolic values are saved for executing path. For input constraints to execute the similar path, it is a must that every statement with decision branch calculates the similar value. The first statement (Line 6 in Fig. 3) will execute as true, because initially the mass is equal to 91, which is more than zero.

(Mass>0)

Now it is the time for a second branch statement which becomes false because volume is automatically set as negative value.

→ (volume>)

With the previous branch statement to form a new path statement:

(Mass>0)^ → (volume>0)

The method fails in the execution of the second condition, so it is altered by negating the branch constraints. When the last condition is negated, then the expression becomes:

(Mass>0) ^ (volume>0)

Now, this new path has passed to a constraint solver to find if there exists an input that executes the new path. There will be many solutions but a tester picks one among all and executes for the next iteration. This time the input can be mass=72 and time=9. Now, it will execute without throwing any exception and returns the type of density i.e. high density. Now, it will execute without throwing any exception and returns the type of density i.e. high density. This path has the following constraints:

(Mass>0) ^ (volume>0) ^ → (density>9) ^ → (density=>9)

Where density= mass/volume. This process continues until stopping criteria is met. This could be possible when the iteration exceeds the sufficient coverage is obtained and threshold value.

3. Related Works

Bokil et al. [2] proposed a tool AutoGen that reduces the cost and effort for test data preparation by automatically generating test data for C code. Auto-Gen takes the C code and a criterion such as state-ment coverage, decision coverage, or Modified Condition / Decision Coverage (MC/DC) as input and considered the small signal generation as the input, it is not covering all the blocks in the model. This belongs to exactly real world, but Zhan considered small Simulink block and Stateflow diagrams are disregarded completely. In our approach we are considering Stateflow models. We have designed Simulink/Stateflow model.

From that model generating the MCDC test data. We are overcoming this limitation of Zhan's approach. Andreas Windisch [5] extended the Zhan's work and in this he generated signal as well as coverage of Stateflow diagrams. In this also one major drawback is it is not applicable to realistic complex models. Tools also there for the testing of Simulink/Stateflow models, one of the tools is Reactive Tester [11], by using guided simulations and heuristics without explanation. This approach is limited regarding the length of generating input signals, model size and complexity leads to lower structural coverage. But our approach overcoming these limitations in our approach works for complex models also generates non-redundant test data that satisfies the specified criterion.

Simulink/Stateflow has originally been designed for the simulation purposes. Automated test generation for Simulink/Stateflow diagram is required to identify the errors. Many authors have tried different ways of test data generation and verification for Simulink/Stateflow diagram. Zhan [4] proposed one approach, novel search based approach to cover the particular structural elements of Simulink. He has Mirko Conard et al. [6] proposed one approach to test suite design for code generation tools. They describe the design of a test suite for code generation tools. This method provides solutions of main problems how the correct transformation of a source into a target language can be proved. The application of the proposed testing approach leads to a test suite which is suitable for testing code generators systematically. Bokil et al. [3] Have presented a tool AutoGen that reduces the cost and effort by automatically generating test data for C code. They have also shown their experience to generate MC/DC test data for three embedded reactive system applications by using AutoGen. They have concluded that the effort required using the tool was one third of the manual effort required.

Gadkari et al. [7] have translated vimulink/Stateflow to a formal language and generated test cases based on model checking. Formal methods are hard to implement. A more mathematical knowledge is required. For small models translating Simulink/Stateflow models of formal language is possible. If the size of the model is increasing translating Simulink/Stateflow to Formal language is difficult. Meng Li and Ratnesh Kumar introduced a recursive method to translate a Simulink/Stateflow diagram to an Input/Output Extended Finite Automata [12] which is a formal model of reactive untimed Input/Output Extended Finite Automata. It is generated manually so it is drawn back.

Pasareanu et al. [17] conduct a survey of new research trends in symbolic execution, with particular emphasis on application to test generation and program analysis. This paper focuses onto the future research

directions of symbolic testing while we focus to generate Branch Coverage test suite from CREST [8] Tool.

There are many Concolic testers like CREST Tool here we discuss the comparative study [16] of the related tools to CREST Tool. Tools identified in the survey, along with their supporting languages, platforms, and underlying constraint solvers. Unfortunately, many of the tools are not available for public use. For instance, DART, CUTE, jCUTE, Path Crawler, and SAGE are all proprietary.

4. Proposed Approach

Our approach Generation of Branch Coverage Test Data for Simulink/Stateflow Models Using CREST Tool mainly consists of three steps:

- **Step1:** We are designing the Simulink model
- **Step2:** We generate the code for the Simulink/Stateowmodel.(Code generation in MATLAB Real-Time Workshop for SL/SF models)
- **Step 3**: Concolic testing for C program code to generate MC/DC test suite

In Step 2 MathWorks Code generation technology generates C or C++ code and executables for algorithms that you model programmatically with MATLAB or graphically in the Simulink environment. You can generate code for any MATLAB functions and Simulink blocks that are useful for real-time or embedded applications. The generated source code and executable for floating-point algorithms match the functional behaviour of MATLAB code execution and Simulink simulations to high degrees of fidelity. Using the Simulink Fixed Point product, we can generate the exact results. The built-in accelerated simulation models in Simulink use code generation technology.

Configuring the model for code generation: To configure the model for code generation we have to set some of the parameters of the model. To do that we open the Configuration Parameters dialog box Solver Pane. Then we set the parameters as mentioned in the table below. After the model configuration parameters are set as specified above, we go to the Real-Time Workshop pane. Generate Code only check box is selected. Generate make file is optional and can be selected if the generated needs to be built directly.

The C program generated from MATLAB is passed to Concolic tester CREST TOOL. The Concolic tester achieves branch coverage through random test generation. Concolic Tester is a combination of concrete and symbolic testing. Test case generation depends on the path of each execution. All test cases stored in text files, which forms as a test suite. The Concolic testing process is carried out using the following six steps:

(a) Symbolic Variables Declaration: It's the starting time; the user has to declare which variable will be symbolic variables so that symbolic path formulas are developed.

(b) Instrumentation: A target source code is statically instrumented with probes, which keeps track of symbolic

(c) Concrete Execution: The instrumented code is compiled and run with given input values. For the values. For the second time onwards, input values are getting from step 6.

First time the target code is assigned with random path conditions from a concrete execution path when the target code is executed. Ex: At each branch, a probe is inserted to track the branch condition.

(d) Symbolic path formula X: The symbolic execution module of the Concolic testing executions collects symbolic path conditions over the symbolic input values at every branch point collides along the concrete execution path. Whenever s statement of the target code is executed, a corresponding probe inserted at 's' updates the symbolic structure of symbolic variables if statement s is an assignment statement, or gathers a corresponding symbolic path condition c, if s is a branch statement. Therefore at last symbolic path formula X is built at the last point of the ith execution by combining all path conditions c1 , c2 , c3 where cj is executed earlier than cj+1 for all 1_j.

(e) Symbolic path formula X′ for the next input values: To find X′ we have to negate one path condition cj and removing after path conditions (i.e. , cj+1, cn) of X′. If X′ is not satisfiable, another path condition cj′ is negated and after path condition are removed, till satisfiable formula is getting. If there are no more paths to try, the algorithm stops executing.

(f) Choosing the next input values: Constraints solver generates a model that satisfies X′. This model takes decision for next concrete input values and this

procedure repeats from step3 again with this input value.

5. Experimental Studies

In this section, we explain the working of our algorithm by taking the Air condition example using a Simulink/Stateflow model, as described below and it is shown in Fig. 3. Fig 3 shows an Air condition Simulink/Stateflow model. It keeps the temperature as constant. In our model we are giving random temperature through Repeating sequence chair. In this chart is a subsystem block which consists of Stateflow model. The Stateflow part of the above example model is shown in Fig. 2. As we have discussed the generation of C code from a model in Step 2 of Section 4, we have generated C code as shown in Fig 8. We feed generated C code in CREST Tool to generate Branch Coverage Test suite. The CREST Tool supports yices constraints solver to test symbolic as well as concrete testing. The ocaml is the compiler for CREST. The compilation of C code using CREST results: a) Number of Read Branches b) Number of Read Nodes c) Number of writing Branch Edges as shown in Fig. 9. The Execution of C code results: a) Number of Covered Branches b) Number of Reachable Functions c) Number of Reachable Branches. There are a number of strategies to execute the C code through CREST like a) DFS, b) CFG, and c) RANDOM.

In our approach we have chosen DFS strategy to execute C code. Here have given iteration number to cover the branches. Initially we provide small number then according to criteria we have increased the iteration number till stopping criteria met. Stopping criteria could be one of the discussed criteria: a) initially we fixed one threshold value, till that value we check beyond that we terminate execution. b) We check till all branches covered, once it done then we have to stop and terminate. At the time of execution CREST saves input values for each iteration which are nothing but test cases or test data as shown in Fig. 10. In Fig. 10 CREST covered 16 branches which show that 16 test cases for 20 iterations, so CREST saves 20 input files as shown in Fig. 11. The CREST int() command used to select input values as shown in Fig. 12 which makes different paths to cover the branches.

6. Conclusion and Future Work

Here we conclude that we can generate Branch Coverage test suite using Concolic tester (CREST Tool) from Simulink/Stateflow model. First we have constructed the model in the MATLAB. Next, by using the simulation we verify the model. After verification, we generated the C code by using the MATLAB Real Time Workshop. This code is executable code. By using Concolic testing we generated the test suite for the C code. This test suite is useful for testing the Simulink/model. We are planning for regression testing of Simulink models using this approach. The future version of this paper will consist of code slicer with code transformer.

Acknowledgment

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

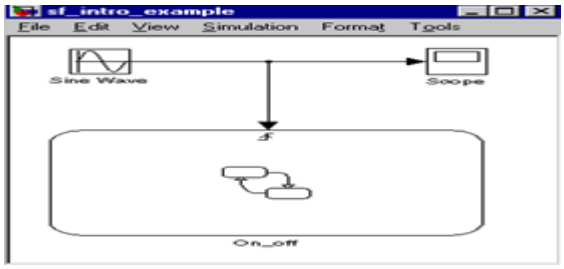

Figure 1: Sample Simulink Example

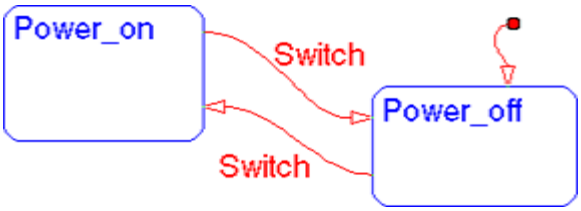

Figure 2: Stateflow model

```
#include<stdio.h>
#include<conio.h>
void main()
{
int density,mass,volume;
if(mass>0)
  continue;
else
   return;
if(volume>0)
  continue;
else
   return;
density=mass/volume;
if(density>9)
   return Low_density;
else if(density==9)
   return Standard_density;
else
   return High_density
}
```

Figure 3: Generated C Code

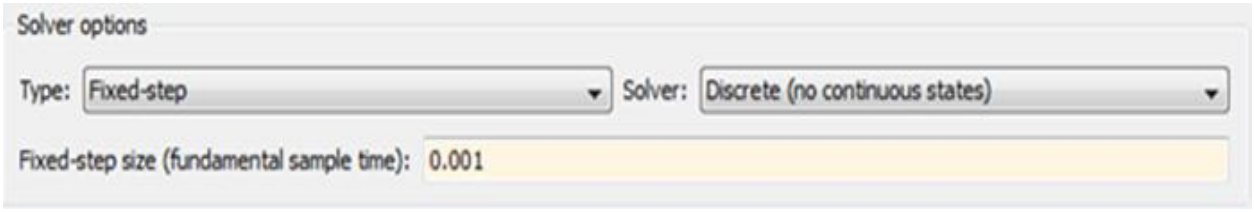

Figure 4: Configuration Parameters Setting Window

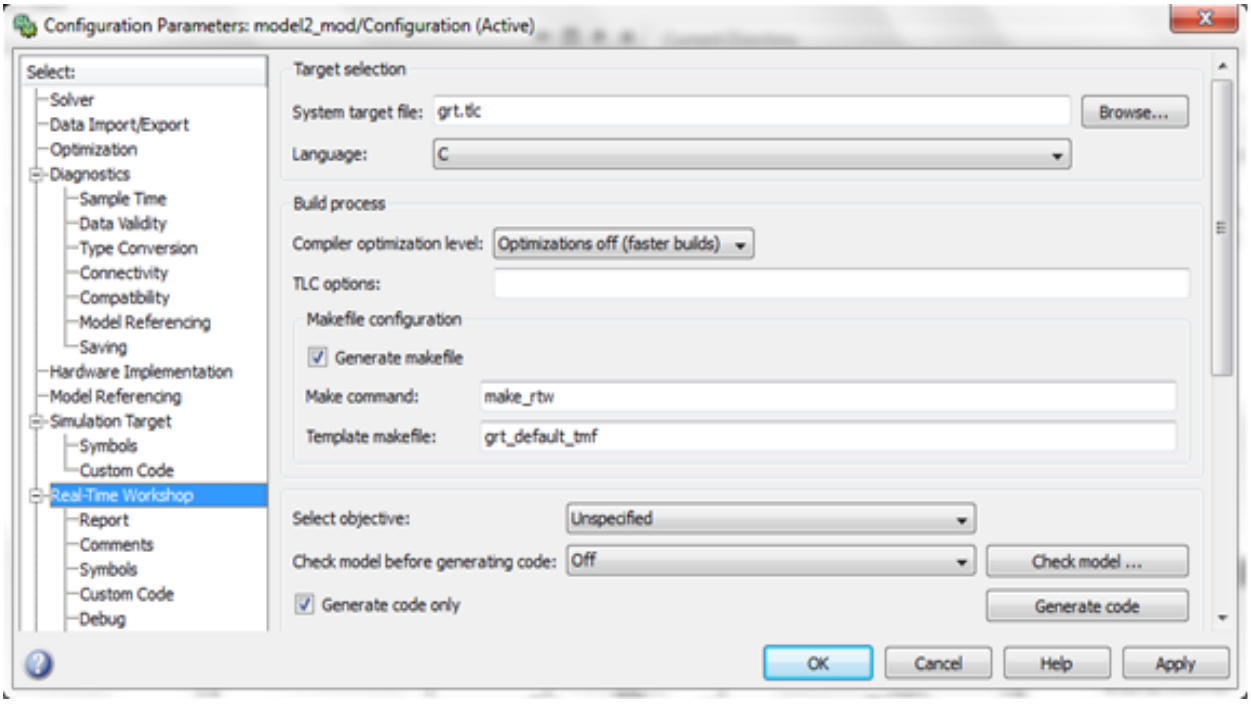

Figure 5: Configuration Parameters

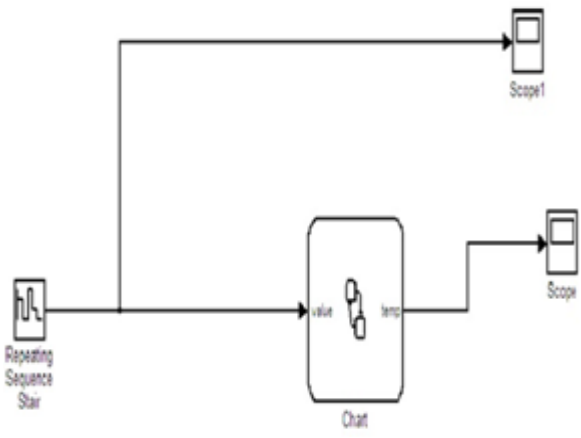

Figure 6: Air Condition Example of Simulink

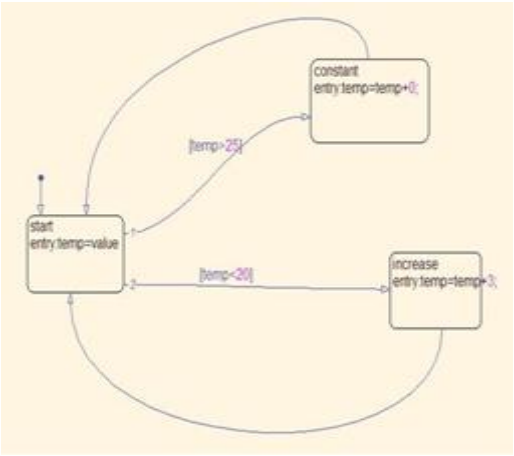

Figure 7: Air Condition Example of Stat

Figure 8: Generated C Code

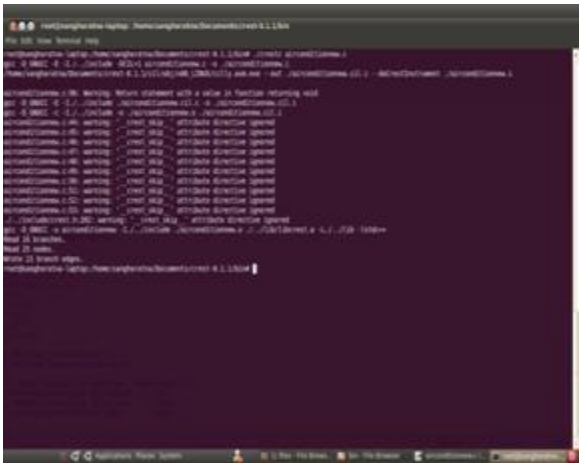

Figure 9: Compilation Results

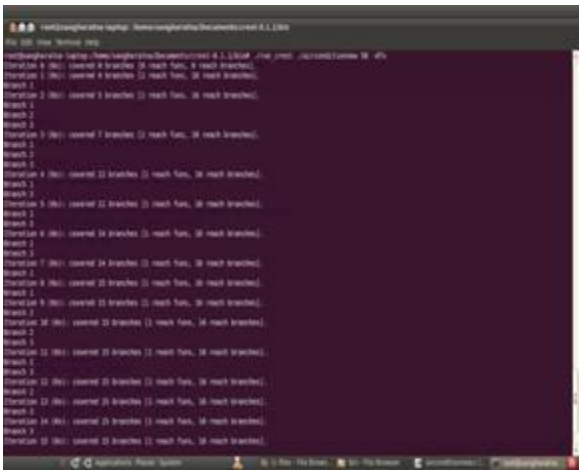

Figure 10: Covered Branches after execution

Figure 11: Automatic generated test suite

References

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

[2] Harel and David. Statecharts: A Visual Formalism for Complex Systems Science of Computer Programming 8 pages 231-274 1987.

[3] Prasad Bokil, Priyanka Darke, Ulka Shrotri and R. Venkatesh. Automatic Test Data Generation for C Programs. In 3rd IEEE International Conference on Secure Software Integration and Reliability Improvement 2009.

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

[5] Andreas Windisch. Search Based Testing of Simulink Models containing Stateflow Diagrams International Conference on Software Engineering - Companion Volume, 2009.

[6] Igno Sturmer and Mirko Conard. Test suite design for code generation Tools. IEEE, 2003.

[7] A. Gadkari, S. Mohalik, K. Shashidhar, A. Yeolekar, J. Suresh, and S. Ramesh. Automatic generation of test-cases using model checking for sl/SF models. 4th International Workshop on Model Driven Engineering, Veri_cation and Validation, 2007.

[8] CREST. http://code.google.com/p/crest.

[9] The Mathworks INC [Online]. http://www.mathworks.com.

[10] Stateflow [Online]. http://dali.feld.cvut.cz/ucebna/matlab/toolbox/Stateflow.

[11] Reactive Systems Inc, Reactis Simulator / Tester. http://www.reactive-systems.com. Website, 2003.

[12] Meng Li and Ratnesh Kumar. Model-Based Automatic Test Generation for Simulink/Stateflow using Extended Finite Automaton. 2011.

[13] J. Chilenski and S. Miller. Application of Modified Condition/Decision Coverage to

Software Testing. Software Engineering Journal, September, 1994, pp. 193-200.

[14] J. C. King. Symbolic execution and program testing. In Commun. ACM, 19 (7), pages 385-394, 1976.

[15] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic unit testing engine for C. In Proc. ESEC/FSE, pages 263-272, Lisbon, Portugal, 2005.

[16] Xiao Qu and Brian Robinson. A Case Study of Concolic Testing Tools and Their Limitations. In Proc. ESEM pages 117-126, Washington, D.C., USA, 2011.

[17] Păsăreanu, Corina S., and Willem Visser. "A survey of new trends in symbolic execution for software testing and analysis." International journal on software tools for technology transfer 11, no. 4 (2009): 339-353.

[18] Godboley, S. Prashanth, G.S. Mohapatra D. P. and Majhi, B. Increase in Modified condition/Decision Coverage Using Program Code Transformer, Feb 2013 In proceedings of 2013 3rd IEEE International Advance Computing Conference (IACC) , Ajay kumarGarg College of Engineering Gaziyabad(U.P), Pages: 1401-1408.

[19] Godboley, S. Prashanth, G.S. Mohapatra D. P. and Majhi, B. Enhanced Modified Condition/Decision Coverage Using Exclusive-NOR Code Transformer, March 2013, In proceedings of 2013 IEEE International Multi Conference on Automation, Computing, Control, Communication and Compressed Sensing (IMAC4S), School of Electronics, St. Joseph's College of Engineering and Technology, Palai, Kottayam, India.

The author's name is Prof. Sangharatna Godboley. He was born on 1st July 1990 at Nagpur (Maharastra). The author completed his B.E degree from Government Engineering College(GEC) Bilaspur, affilited with CSVTU Bhilai University. He did his M.Tech degree from National Institute of Technology(NIT) Rourkela under the kind guidance of Prof. D. P Mohapatra and Prof. B. Majhi. He is an IEEE Student Member and an IEEE Communication Society member since 1st Jan 2013. He has published two IEEE International Conferences, one Springer Conference, one 7th CONSEG Conference, and one ICETTR Conference IJACR. He communicated with two more Research Papers.

The author's name is Prof. Adepu Sridhar and was born on 16th June 1990 at Karimnagar Andhra Pradesh. The author completed his B.Tech Degree from University College of Engineering (Kakatiya University). He did his M.Tech degree from National Institute of Technology Rourkela under the guidance of Prof. D. P Mohapatra . He is an IEEE student member since1st Jan 2013. He has published one IEEE international conferences and one Springer conference. He communicated with four more research paper.

The author name is Bhupendra Kharpuse and was born on 17th July 1989 at Chhindwara Madhya Pradesh. The author completed his B.E degree from Malwa Institute of Technology Indore, affiliated with RGPV Bhopal University. He did his M.Tech degree from Indian Institute of Technology Kharagpur under the guidance of Prof. Rajib Mall. Currently he is working in Symantec India Pvt. Ltd Pune.

Prof. Durga Prasad Mohapatra received his Ph. D. from Indian Institute of Technology Kharagpur and M. E. from Regional Engineering College (now NIT), Rourkela. He joined the faculty of the Department of Computer Science and Engineering at the National Institute of Technology, Rourkela in 1996, where he is now Associate Professor. His research interests include software engineering, real-time systems, discrete mathematics and distributed computing and published more than forty papers in these fields. He has received many awards including Young Scientist Award for the year 2006 by Orissa Bigyan Academy, Prof. K. Arumugam award for innovative research for the year 2009 and Maharasthra State National Award for outstanding research for the year 2010 by ISTE, New Delhi. He has also received three research projects from DST and UGC. Currently, he is a member of IEEE. Dr. Mohapatra has co-authored the book Elements of Discrete Mathematics: A Computer Oriented Approach published by Tata Mc-GrawHill.Computer Science and Engineering Dept., National Institute of Technology, Rourkela, India.

Prof. Bansidhar Majhi received his Ph. D and M. E. from Regional Engineering College (now NIT), Rourkela. He joined the faculty of the Department of Computer Science and Engineering at the National Institute of Technology, Rourkela in 1991, where he is now Professor. His research interests include Soft Computing, Image processing, Biometrics, Security Protocols. He is a member of professional bodies likeFIETE, LMCSI, and AMIE (INDIA). Computer Science and Engineering Dept.,National Institute of Technology, Rourkela, India.