MODEL REDUCTION ALGORITHM FOR FAST NEUTRALITY TESTS AND FAULT LOCALIZATION OF SIMULINK MODELS

X. Hu, A. Morozov[1], K. Ding, K. Janschek[1], N. Yusupova[2]

Technische Universität Dresden, Germany[1]
Faculty of Computer Science and Robotics, Ufa State Aviation Technical University, Russia[2]

xinrui.hu tu-dresden.de, andrey.morozov tu-dresden.de, kai.ding tu-dresden.de, klaus.janschek@tu-dresden.de, yussupova@ugatu.ac.ru

Abstrac:t *A minor change of a Simulink model can result in an unexpected consequence, so the Simulink model is usually required to be rerun and tested, which increases the development cost and time. Compared with the reference model, only the changed parts of the updated model could result in a failure at the outputs. So, a two-stage model reduction algorithm is designed to isolate the changed parts, that speeds up the processes of neutrality test and fault localization. The first reduction is based on the changed parts, the second reduction is based on the bad outputs. The changed parts and the bad outputs are the blocks of interest of the reduction. The blocks related to the blocks of interest are reserved, the others are deleted. The thesis proposes a way of conversion of the Simulink model to a digraph based on extended data dependence to find the related blocks. After the model reduction, the faults are located with the help of signal comparison.*

1. Introduction

The methodology of Model-Based Development is widely used in modern mechatronic and cyber-physical systems of higher heterogeneity and larger scale. One of the world-wide recognized tools is MATLAB/Simulink, a common tool for the system- and component-level design and simulation, automatic code generation and its deployment on target hardware [13], which allows the engineer to verify it early in the life cycle.

Even after minor changes, the reliability standards require that the Simulink model should be tested to ensure no fault is introduced. One of the model test activities is the fault localization. The commonly used method, regression test, is often used to ensure that there aren't any newly introduced faults after the update of the model. It often requires rerunning the whole model and the whole test cases, which consumes an amount of time. In order to decrease the costs of the fault localization, the researchers have investigated a lot of useful ways like statistical debugging [1, 2, 7, 10], Simulink Design Verifier [26], Falsification-based testing using Temporal Logic [3, 5].

However, it is assumed that only the changed blocks of the Simulink model may be faulty. In contrast with rerunning the entire updated model, the thesis designs the two-stage of model reduction algorithm to reduce the scale of the models and the faults are located based on the reduced models.

2. State of the Art

Simulink/Simulink API: As the world-wide most recognized tool for model-based system design, MATLAB/Simulink [13] is widely used in industrial application. Simulink and Simulink Model-Based Design aim at helping to implement the system with high quality, especially for the extremely complicated systems. It allows developers to design, analyze and simulate the models before moving to the real hardware world. After the model test, the codes, e.g. C, C++, can be generated from those verified models automatically and deployed on target hardware. The Simulink model is a network of blocks of different types. By drawing blocks and lines, the designer can have experience of intuitive graphical design. The configuration of the blocks and models can be implemented manually by a user interface or programmatically by Simulink API. Simulink provides different types of solvers [16], such as fixed-step and variable-step solvers, to simulate dynamic systems for different requirements.

Simulink model reduction: Simulink Model Slicing is widely used in many approaches of faults localization [5, 9]. Ways of model slicing can be basically classified into two groups: the official tool - Simulink Model Slicer provided by Simulink [14], which also allows users to get access programmatically using Simulink API, and the unofficial tool based on dependence analysis [12]. The former isolates problematic behavior in a model and allows the user to highlight and trace ports, signals, and blocks, and then slices a large model into smaller models for further analysis [14]. The latter slices the Simulink models by using a dependence graph, which is calculated by analysis of the data dependence and the control dependence, so that the models can be reduced for the given blocks of interest.

Fault localization for Simulink model: So far there are lots of investigation to fault localization for Simulink models. They can be roughly classified into two categories: official tool from Simulink - Simulink Design Verifier (SDV) [14] and unofficial tools. The unofficial tools are mainly developed based on the statistical debugging method and falsification testing using temporal logic. Statistical debugging is a well-studied and wide-used debugging technique in software engineering domain [1, 2, 7]. The statistical debugging is extended to fault localization for the Simulink model and search-based test suite generation algorithm are proposed in [9].

The other tool is based on Signal Temporal Logic (STL). STL is originally proposed for hardware circuits. With growing interest in model checking for more sophisticated fields such as analog/digital mixed signal circuits, cyber-physical systems this approach has been also widely used in fault localization for Simulink/Stateflow models [5]. It uses execution-context-based model slicing [12] and finds the faults that result in a violation of STL at the outputs of the sliced model.

SDV is a powerful product of Simulink, which includes abundant products such as Simulink Requirements, Simulink Check, and Simulink Coverage, aiming at generating test cases based on design requirements, detecting basic modeling errors such as dead logic, integer, and fixed-point overflows, division by zero, etc. With help of Model Slicer tool in SDV, the potentially problematic parts of a time interval of interest will be isolated. But basically, the user should define an interval of interest manually. Usually after the first model slicing the experienced user need to adjust the interval in order to find the problematic position more precisely. This is hence an iterative process.

Neutrality test and regression test: Inspired by the neutrality theory of biology domain, modifications or changes which do not affect the system are neutral. Neutrality testing aims at determining whether changes in model interfere the parts of interest by using neutrality as a null hypothesis [6]. In the software engineering domain, even a small tweak can result in unexpected consequences. The regression test is a test method to make sure that changes don't

cause unintended effects. Usually, regression test requires rerunning test cases and check if violations at observed position appear. Due to the large scale of test suits regression test is a quite time-consuming progress. So many techniques have been proposed to make the regression test more cost-efficient. They can be mainly classified into three groups [17]: test suite minimization [15], test case selection [11] and test case prioritization [8].

3. Design and Implementation

In order to reduce the validation time of an updated or modified model, the model is reduced two times. After two reductions, the faults will be located. The first stage of model reduction is based on the changed parts and the blocks related to the changed parts. After the first reduction, the models are executed to find the bad outputs. The second reduction is based on the bad outputs and the blocks related to the bad outputs. The conversion of the Simulink model to the digraph is designed. With the help of the digraph and the corresponding adjacency matrix, signals are traced and the related blocks are found. After two reductions, the scale of the model is smaller. Based on signal comparisons of the reserved blocks, faults are located. Figure 1 shows the algorithm of the two-stage model reduction for fault localization.

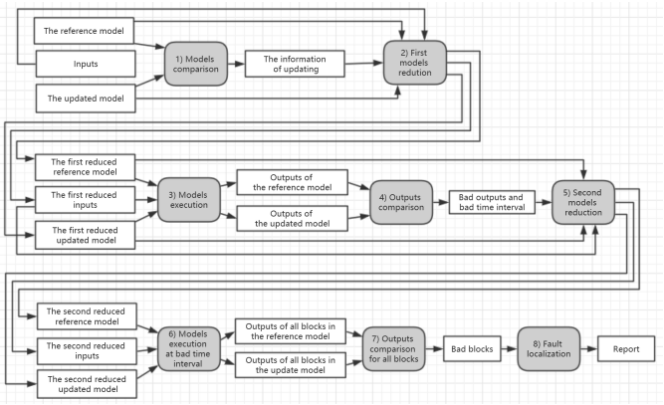

Fig. 1 *Algorithm of the model reduction and fault localization*

3. 1. Signal Tracing and Model Reduction

a) Conversion of the Simulink model to the digraph: The two model reductions have the similar idea, i.e. to reduce the models to the given blocks of interest. The blocks related to the blocks of interest are found and reserved, and the other blocks and redundant lines are deleted. To find the related blocks, the Simulink model is first converted to the digraph. It is defined that, the nodes in the Simulink model is a vertex in the digraph. The signal transmission is an arc in the digraph. To make sure that the models can be executed normally, the signal transmission is defined as the follows: i) The directed lines in the Simulink transmit signals from the start to end of the line. ii) The *Goto* block transmits signals to *From* block with the same *GotoTag*. iii) The blocks, connected with the condition port of the conditionally execution subsystems (the *triggered*, *enabled*, and *action* subsystem), transmits signals to the *trigger*, *enable*, and *action* port inside the subsystems. iv) The *trigger*, *enable*, and *action* port transmit signals to all blocks inside the conditionally execution subsystems.

In graph theory, directed graph (or just digraph) D consists of a non-empty finite set V (D) and finite set A(D), where [4],

- V(D) (vertex set) is a finite set of vertices. It is denoted as $V(D) = v_1,\ v_2 \ldots, v_n\ \}$.
- A(D) (arc set) is a finite set of ordered pairs of distinct vertices namely arcs. It is denoted as $A(D) \coloneqq \{(v_i, v_j) |\ v_i, v_j \in V(D)\}$.

An example in figure 2 shows the conversion of the Simulink model to the digraph.

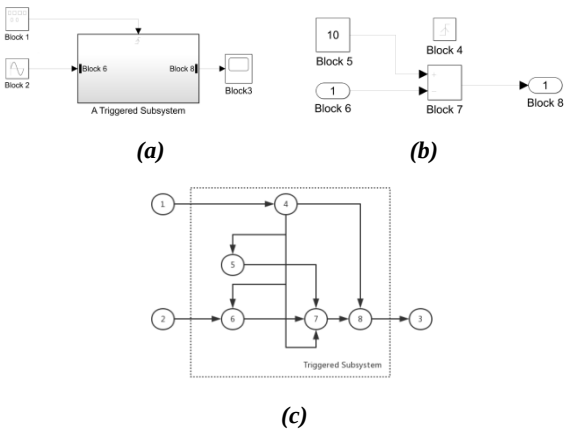

Fig. 2 *An example of the transformation of the Simulink model to digraph: (a) Simulink model with triggered subsystem (b) Internal structure of triggered subsystem (c) Digraph of the Simulink model*

b) Signal tracing based on the digraph: As explained, the nodes in the digraph represent the blocks in the Simulink model. In order to find the related blocks of the block interest, the adjacency matrix of the digraph is used. Based on the digraph, the related nodes of the node of the interest are defined in three groups:

1. The nodes that are backward-connected with the node of interest.
2. The blocks that are forward-connected with the block of interest.
3. The blocks that are backward-connected with the blocks from the group 2.

Figure 3 shows an example of a digraph of a Simulink model. Assume that the node 4 is the node of interest. The related nodes from group 1 are nodes 1, 2, 3; the related nodes from group 2 are nodes 5, 8; the related node from group 3 is node 6. Only node 7 is note related to node 4. After the model reduction, node 7 is deleted.

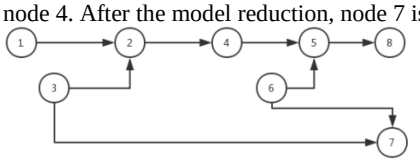

Fig. 3 *An example digraph for signal tracing*

3. 2. Signal Comparison

Simulink provides several ways to get the simulation data, such as *Simulink.sdi.sim*. The former provides an interface to get access to Simulation Data Inspector [18] programmatically. It helps to access Run and Signal objects in Simulation Data Inspector by using functions *Simulink.sdi.getRun* and *Simulink.sdi.getSignal* respectively. However, the inputs of the functions are *runID* and *sigID,* meaning the ID number of the signal object and the simulation run object, which make it quite difficult to distinguish a *sigID* represents which signal object and a *runID* represents which run object. The latter is a function to simulate a model programmatically. Using sim makes it possible to specify parameter name-value pairs, to configurate simulation, to access simulation metadata, etc.. The parameter name-value pairs can be specified before simulation, meaning that the signals, which will be compared, can be pre-set names so that they can be accessed by names. Based on this reason, sim will be used for getting the simulation data. *Simulink.SimulationData.Signal* stores simulation data, it has a Values property. The Values field represents a timeseries type of variable. It contains data variable and the corresponding time variable. Signal comparison is based on these two variables. Two signals will be considered as "bad" if any one of the situations occurs: a) time variables aren't the same, b) time variables are the same, but the data variables differ.

3. 3. Fault Localization

Based on the signal comparison, the models will be run only at the "bad" sample time intervals, which saves a lot of time. It might happen that some "bad" time intervals are too long, the upper limits of "bad" time interval will be given. The inputs of the reference and updated models will also be reduced to the given "bad" time interval. However, in a Simulink model, there are some delay. For example, assume the simulation step time is T, if there is a block b_2 which receive signal from block b_1, b_2 is a delay block, which delays the signal e.g. 1T. So if b_1 outputs a fault at time point kT, b_2 will output the fault however at the time point $(k+1)T$. The outputs of the blocks forward-connected with b_2 will output the fault at (k+1)T or even later. In this case, the bad time interval shouldn't start at time point kT, otherwise, it could happen that after reduction of the inputs, the fault will disappear. So, the margin time should be taken into account when generating the bad time intervals. Noted that the calculation process of the Simulink block doesn't use the value at the future point, the detected error at the output is only dependent on the current time point or earlier time points, thus, the margin time should be added at the front of the bad time interval.

After finding the bad time intervals, the model will be run to perform fault localization. Each non-subsystem blocks in Simulink model is a system element, in our thesis called blocks b. Each element b_i has inputs and outputs, defined as [17]:

- $I_{b_i} := \{i_1, i_2, \dots, i_{N_{b_i}^{in}}\}$,
- $O_{b_i} := \{o_1, o_2, \dots, o_{N_{b_i}^{out}}\}$

, where I_{b_i} and O_{b_i} are the sets of inputs and outputs of the block b_i, $N_{b_i}^{in}$ and $N_{b_i}^{out}$ are numbers of inputs and outputs of b_i. Then the faulty block b_f satisfies the following criteria:

- It is a related block of the bad outputs.
- If it has inputs, and all the inputs are correct (equal to the reference), but not every output is correct.
- If it doesn't have input and not every output is correct.
- If it is an added or deleted block, it is considered as potentially faulty.

Figure 4 shows the example of the faulty blocks. The red colored lines represent the incorrect signals, the black colored are correct.

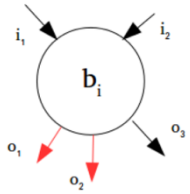

***Fig. 4** An example of a faulty block*

In Simulink, it's not allowed to log signals, such as *action* signals, signals that feed into a *Merge* block, *function-call* signals, or *sate* signals. This thesis doesn't handle with the models containing *function-call* signals and *sate* signals, Also, there are some blocks don't have real data signal lines, and some blocks' lines are control signal lines.

For example:

1. *Inports* in a subsystem don't have inputs signal lines;
2. *Outports* in a subsystem don't have outputs signal lines;
3. If and *SwitchCase* blocks only have control or more explicitly action output signals.
4. The *Action* ports in *Action* subsystems have neither inputs nor outputs.
5. The *Trigger* ports in *Triggered* or *Function-Call* subsystems have neither inputs nor outputs.
6. The *Enable* ports in *Enabled* subsystems have neither inputs nor outputs.

The ports mentioned in situations 4, 5, and 6 accept the signals to start the execution of conditional execution context [12]. The process is done by the internal schema of Simulink, it can be considered as always correct. So, these special ports won't be checked. The outputs of *If* and *SwitchCase* are *action* signals so that their outputs can't be logged directly. However, the Action subsystems which are predicted by them can give some information. By logging the output value of *Inports* blocks, the activation time points of the subsystems can be obtained. It gives information about if an action signal is given. So, *If* or *SwitchCase* blocks are blocks,

- whose inputs are all correct (equal to the reference),
- and the activation time points of the subsystems connected with its *outports* are incorrect.

4. Results and Discussion

To validate the program, a model with lookup tables in figure 5 is chosen. Since one of the inputs of the 2-D lookup tables is constant, which means some breakpoints may not be used. One breakpoint of *Pumping Constant*, which is not used, is modified. One breakpoint of *Ramp Rate Ki*, which is used, is modified. In addition, two blocks are deleted in the updated model.

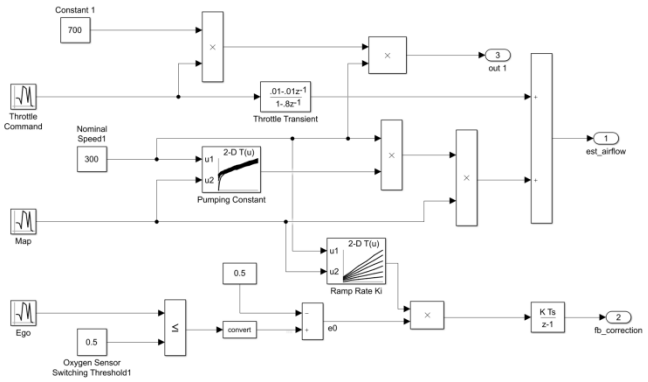

***Fig. 5** A validation model with lookup tables*

After the execution of the program the faults are located, and the reduced model is in figure 6, the report of fault localization is in figure 7. Table 1 shows the number of the blocks after each model reduction.

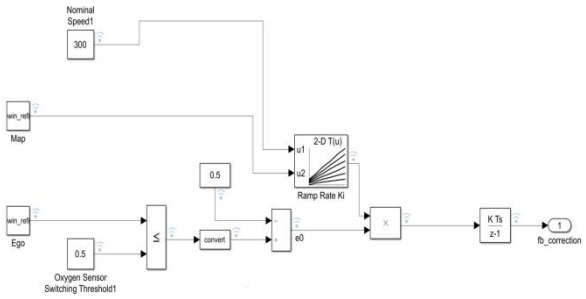

***Fig. 6** The reference model after two reductions*

```
>> fault_localization
The faults may locate at the following blocks:
1. RampRateKi
2. deleted Sum1
3. deleted Constant
```

***Fig. 7** The report of fault localization*

From this example, it can be seen that, the models are first reduced to the changed parts. After the first reduction, the models are run. The first output *est_airflow* of the model only receives the signal from *Pumping Constant*, the second output *fb_connection* only receives the signal from other changed parts. However, the changed breakpoint in the lookup table *Pumping Constant* is never used, no error is produced and propagated to the first output, so the first output is correct. The second reduction is based only on the second output. After two reductions, the scale of

the model is obviously reduced. The deleted blocks are related to the bad outputs, so they are considered as potentially faulty.

Table 1: *Numbers of blocks after each model reduction*

Stage	Number of blocks in the reference model	Number of blocks in the updated model
Before reduction	23	21
After first reduction	19	17
After second reduction	12	10

5. Conclusion

Since the Simulink model is modelled by drawing the visible and intuitive lines and blocks, so the idea of doing fault localization of updated models based on signal tracing and signal comparison comes up naturally. In addition, faults can only be located at the positions, where they are not the same as a previously validated model. So, the updated model will be compared with the previously validated model, or the reference model. Hence, the neutrality testing based on model reduction is proposed.

The work consists of three main tasks: model reduction, signal tracing, and signal comparison. A controller model contains normally two parts: control logic and data signals. The latter can be analyzed along the lines in the model. The former however is an internal mechanism, which isn't so intuitive. The built-in Simulink Model Comparison tool provides a way of model comparison. However, it doesn't give information about which blocks use the modified variables in the model workspace, which should be solved additionally. After model comparison, the model is reduced to a new model with only those blocks which are relevant to the modified blocks. Hence, an approach of transforming of the Simulink model to digraph is proposed to solve the problem of signal tracing and finding the related blocks.

Furthermore, in order to do fault localization, the signal tracing and signal comparison are needed to find which blocks are guilty for the detected bad outputs of a model. With the help of signal comparison and the adjacency matrix of the digraph. The fault blocks can be found.

However, the project has some limitations that will be solved further. i) The comparison of the two models is based on the names of the blocks. If the names of blocks are changed or added with an unintentional whitespace symbol, then they will be reported as faulty blocks. ii) If there are faults in two cascaded connected blocks, each execution of the program can detect only the first faulty block. However, if the program is executed iteratively, the second faulty block can be found. iii) If several faulty blocks result in bad signals at the same output one after the other, then only the block, which produces the first error, will be detected. This can be also solved by the interactive execution of the program. iv) The program can’t locate faults for the model with S-Function and Stateflow Chart.

Reference

[1] James A. Jones and Mary Harrold. "Empirical evaluation of the Tarantula automatic fault-localization technique". In: 20th IEEE/ACM International Conference on Automated Software Engineering, ASE 2005. Jan. 2005, pp. 273-282.

[2] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. "On the Accuracy of Spectrum-based Fault Localization". In: Testing: Academic and Industrial Conference Practice and Research Techniques - MU- TATION (TAICPART-MUTATION 2007). Sept. 2007, pp. 89- 98.

[3] Yashwanth Annpureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. "S-TaLiRo: A Tool for Temporal Logic Falsification for Hybrid Systems". In: Tools and Algorithms for the Construction and Analysis of Systems. Ed. by Parosh Aziz Abdulla and K. Rustan M. Leino. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 254-257.

[4] Jrgen Bang-Jensen and Gregory Z. Gutin. "Basic Terminology, Notation and Results". In: Digraphs: Theory, Algorithms and Applications. Springer Publishing Company, Incorporated, 2008. Chap. 1.

[5] Ezio Bartocci, Thomas Ferr_ere, Niveditha Manjunath, and Dejan Nickovic. "Localizing Faults in Simulink/Stateflow Models with STL". In: Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (Part of CPS Week). HSCC '18. Porto, Portugal: ACM, 2018, pp. 197-206.

[6] Yun-xin Fu. "New Statistical Tests of Neutrality for DNA Samples From a Population". In: Genetics 143.1 (1996), pp. 557-570.

[7] J. A. Jones, M. J. Harrold, and J. Stasko. "Visualization of test information to assist fault localization". In: Proceedings of the 24th International Conference on Software Engineering. ICSE 2002. May 2002, pp. 467-477.

[8] Muhammad Khatibsyarbini, Mohd Adham Isa, Dayang N.A. Jawawi, and Rooster Tumeng. "Test case prioritization approaches in regression testing: A systematic literature review". In: Information and Software Technology 93 (2018), pp. 74-93.

[9] B. Liu, Lucia, S. Nejati, and L. C. Briand. \Improving fault localization for Simulink models using search-based testing and prediction models". In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). Feb. 2017, pp. 359-370.

[10] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midki. "SOBER: statistical model-based bug localization". In: ESEC/SIG-SOFT FSE. 2005.

[11] Everton Note Narciso, Marcio Delamaro, and Fatima De Lourdes Dos Santos Nunes. "Test Case Selection: A Systematic Literature Review". In: 24 (May 2014), pp. 653-676

[12] R. Reicherdt and S. Glesner. "Slicing MATLAB Simulink models". In: 2012 34th International Conference on Software Engineering (ICSE). June 2012, pp. 551-561.

[13] Simulation and Model-Based Design. https://www.mathworks.com/products/simulink.html. Access: 2018-08-30.

[14] Simulink Design Verifier. https://www.mathworks.com/products/sldesignverifier.html Accessed: 2018-08-30.

[15] Rajvir Singh and Mamta Santosh. "Test Case Minimization Techniques: A Review". In: 2 (Dec. 2013), pp. 1048-1056.

[16] Simulink Solvers. https://www.mathworks.com/help/simulink/ug/solvers.html Accessed: 2018-08-30.

[17] S. Yoo and M. Harman. "Regression Testing Minimization, Selection and Prioritization: A Survey". In: Softw. Test. Verif. Reliab. 22.2 (Mar.2012), pp. 67-120.

[18] Simulation Data Inspector in Your Workflow. https://www.mathworks.com/help/simulink/ug/simulation-data-inspector-overview.html. Accessed: 2018-07-12