DISSERTATION

Presented on 13/07/2017 in Luxembourg
to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG
EN INFORMATIQUE

by

BING LIU
Born on 20 April 1987 in Xingping (Shaanxi, China)

AUTOMATED DEBUGGING AND FAULT LOCALIZATION OF MATLAB/SIMULINK MODELS

Nothing in the world can take the place of persistence. Talent will not: nothing is more common than unsuccessful men with talent. Genius will not: unrewarded genius is almost a proveb. Education alone will not: the world is full of educated derelicts. Persistence and determination alone are omnipotent.

The 30th President of the United States
CALVIN COOLIDGE

Going in one more round when you don't think you can, that's what makes all the difference in your life.

ROCKY BALBOA

Abstract

Context. Matlab/Simulink is an advanced environment for modeling and simulating multidomain dynamic systems. It has been widely used to model advanced Cyber-Physical Systems, e.g. in the automotive or avionics industry. To ensure the reliability of Simulink models (i.e., ensuring that they are free of faults), these models are subject to extensive testing to verify the logic and behavior of software modules developed in the models. Due to the complex structure of Simulink models, finding root causes of failures (i.e., faults) is an expensive and time-consuming task. Therefore, there is a high demand for automatic fault localization techniques that can help engineers to locate faults in Simulink models with less human intervention. This demand leads to the proposal and development of various approaches and techniques that are able to automatically locate faults in Simulink models.

Fault localization has been an active research area that focuses on developing automated techniques to support software debugging. Although there have been many techniques proposed to localize faults in programs, there has not been much research on fault localization for Simulink models. In this dissertation, we investigate and develop a lightweight fault localization approach to automatically and accurately locate faults in Simulink models. To enhance the usability of our approach, we also develop a stand-alone desktop application that provides engineers with a usable interface to facilitate localization of faults in their models.

Approach. In this dissertation, we propose a set of approaches based on statistical debugging and dynamic model slicing to automatically localize faults in Simulink models. We further propose techniques to improve fault localization for Simulink models by means of generating test cases based on meta-heuristic search and machine learning techniques. The work presented in this dissertation is motivated by Simulink fault localization needs at Delphi Automotive Systems, a world leading part supplier to the automotive industry.

In this dissertation, we propose the following techniques for fault localization of Simulink models: (1) We propose a fault localization approach for Simulink models by extending the statistical debugging technique to Simulink models. We use dynamic slicing to generate Simulink model spectra such that each spectrum is related to one model output and one test case. We then apply statistical ranking formulas to the resulting spectra to compute suspiciousness scores for each Simulink model block. We also propose an iterative fault localization algorithm to further improve fault localization accuracy and a heuristic stopping criterion to avoid unnecessary expansion of test oracles. (2) We investigate the performance of our fault localization technique when it is applied to Simulink models containing multiple faults. We provide mechanisms to help engineers localize faults effectively when the underlying Simulink models contain more than one fault. Specifically, we propose an iterative approach to handle multiple faults using a supervised learning technique (decision trees). The decision trees are used to divide the test coverage data related to the failing test cases into a number of sets such that each set is likely to relate to an individual fault in the model. Each set gives rise to a single ranking. We provide engineers with criteria to select among these rankings the one in which the faulty block is more likely to be ranked high in the list. Once a fault is found and corrected, engineers can reapply our approach to find further faults. (3) Finally, we provide a test suite generation algorithm to improve fault localization of Simulink models by generating small and diverse test suites, and also we develop a strategy to stop test generation when it is unlikely to improve fault localization.

Our fault localization techniques are evaluated based industrial subjects (Simulink models from the automotive industry).

Contributions. The main research contributions in this dissertation are:

1. An approach to localize faults in Simulink models (assuming that they contain a single fault) based on statistical debugging and dynamic slicing. Our approach is the first to extend statistical debugging to Simulink models.
2. A new technique to localize faults in Simulink models containing multiple faults using a supervised learning technique (decision trees learning). Our approach builds on statistical debugging and is iterative.
3. A search-based testing technique for Simulink models that uses three existing alternative test objectives to generate small and diverse test suites that can help improve fault localization accuracy.
4. A Simulink Fault Localization tool (*SimFL*) implementing our fault localization techniques. Our tool provides a usable interface allowing engineers to interact and reiterate our techniques on Simulink models.

Acknowledgements

I would like to thank all the people who helped me, encouraged me, supported me, and accompanied me throughout the past four years of my Ph.D. studies.

First and foremost, I want to give my deepest thanks to my Ph.D. supervisor, Prof. Lionel Briand, who has offered me such a great opportunity to work and pursue my Ph.D. degree under his supervision and spent tremendous amount of time and effort to help me with various problems that I faced in my research as well as daily life. I am grateful to have such a chance to work with and learn from one of the top researchers in software engineering field.

Secondly, I would also like to express my gratitude and special thanks to my co-supervisor, Dr. Shiva Nejati, for all her encouragement, patience, and guidance throughout my Ph.D. study. She has taught me and given me a lot of valuable advice on how to perform research, how to organize and manage research ideas, how to write good scientific papers, how to make professional presentations, and etc. At the same time, she also tried her best to train me to be an independent researcher. I will always remember the period that we worked side by side on my first paper. Even though publishing this paper was the most difficult and tough time during my entire Ph.D. study, I did not lose faith in myself and I believe that was her encouragements that gave me strength. I am very grateful to have such a mentor.

Thirdly, I am very much thankful to two great research collaborators: Dr. Stephan Arlt and Dr. Lucia, who have significantly helped me with the work presented in this dissertation. They also helped and supported me in both my research and daily life. I will remember all the insightful discussion with them in different periods of my Ph.D. study. I am grateful to have such two friends and collaborators.

I would also like to express my gratitude to our industrial partner, Delphi Automotive System, for their help and support. In particular, I would like to thank Thomas Bruckmann, Claude Poull and Camile Feyder for providing me with numerous technical assistance and discussion, in which I learnt and understood a lot of detailed information about Delphi systems.

I am so proud to be a member of the Software Verification and Validation Lab in the Interdisciplinary centre for Security, Reliability and Trust (SnT) at the University of Luxembourg. It is my great pleasure to work with all the very brilliant peers in the SVV group. Thanks for their constant support, interesting discussions and useful advice on my research.

Last but not least, I am very grateful to my family, My dear wife Yawei Zhao and my little daughter Luoran Liu. Both of them provide me their continuous support, encouragement, and love. Without their support, this dissertation would not have been possible. I would also like to thank my parents and sister for their lifelong support and for enabling me to pursue my dreams.

Fonds National de la Recherche Luxembourg DELPHI

Supported by Delphi Automotive Systems and the Fonds National de la Recherche, Luxembourg (FNR/P10/03 and FNR 8003491).

Contents

List of Figures

List of Tables

Acronyms

API	Application Programming Interface.
CPS	Cyber-Physical Systems.
ECUs	Electrical Control Units.
FNR	Fond National de la Recherche.
HC	Hill-Climbing.
HCRR	Hill-Climbing with Random Restarts.
HiL	Hardware-in-the-Loop.
HPC	High Performance Computing Platform.
SimFL	Simulink Fault Localization.
SUT	System Under Test.

Chapter 1

Introduction

1.1 Context

Nowadays, software systems appear in all aspects of our society. However, every coin has two sides. On one hand, the pervasion of software systems makes people's lives more convenient. On the other hand, ensuring the reliability and robustness of software systems is increasingly a difficult and critical task. For example, the first significant study about the cost of software failures was published in 2002 by NIST [National Institute of Standards and Technology U.S. Department of Commerce, 2017]. In this study, it is reported that "Software bugs are costing the U.S. economy an estimated $59.5 billion each year", and improvement in testing and debugging could reduce this cost by about one third ($22.5 billion). This was the first time that the cost of software bugs was precisely measured nationwide. Since then, more and more developers and engineers have realized that software errors are non-negligible and may lead to high cost and overhead. In the following years, researchers spent a lot of effort working on the topic of testing and debugging software systems. However, due to the explosive growth of software products, applications and e-services in recent years, the threats from software defects have become even more acute. Recently, another disturbing software failure review has been released by Tricentis [TRICENTIS, 2017]. It reports that 4.4 billion people and 1.1 trillion assets were affected globally in 2016 by software failures. The quality of software systems increasingly impacts everyone's life. Particularly, the quality of safety critical software systems in domains such as automotive and avionics, whose failure could result in catastrophic consequences, is becoming increasingly crucial.

This dissertation presents a set of approaches based on statistical debugging and machine learning techniques to automate localizing faults in Simulink models in a cost-effective and practical manner. The work presented in this dissertation has been done in collaboration with Delphi Automotive Systems [Delphi Automotive LLP, 2017], a world leading automotive supplier, based in Luxembourg.

1.2 Thesis Background

The automotive software industry increasingly relies on Simulink to develop embedded software components [Thums and Quante, 2012, Reicherdt and Glesner, 2012, Sridhar and Srinivasulu, 2014]. The Simulink language, being supported by advanced automated code generators, has become a prevalent

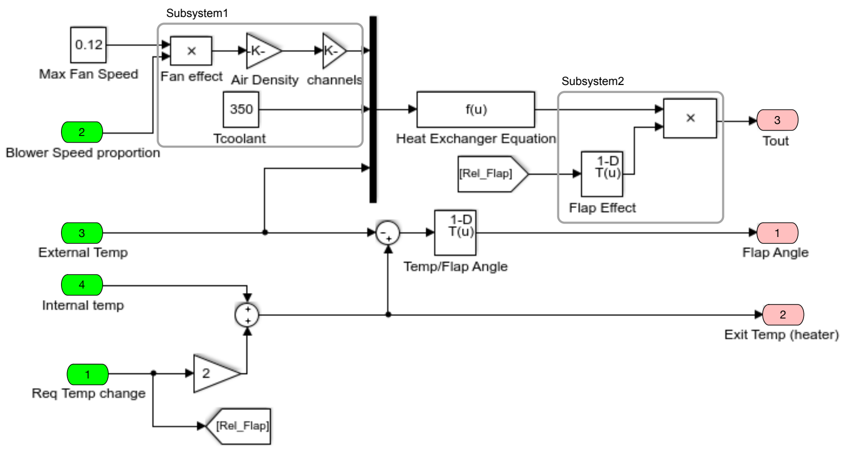


Figure 1.1. A simple Simulink model from Mathworks (Vehicle Heater control system).

language for implementing embedded software. These days nearly every automotive software module is first developed as Simulink models from which C code is later generated automatically. These Simulink models are subject to extensive testing and debugging before code generation takes place. Testing Simulink models is the primary testing phase focused on verification of the logic and behavior of automotive software modules. Furthermore, Simulink model testing is more likely to achieve a high level of fault finding compared to testing code as Simulink models are more abstract and more understandable by engineers. Given the importance of testing and debugging Simulink models, an automated technique to localize faults in Simulink models is crucial.

As shown in Figure 1.1, Simulink models are visual and data-flow-based, and consist of blocks and lines. Each block performs a certain operation on a set of input signals and produces a set of output signals, whereas each line connects two blocks to establish a flow of data between them. Simulink models often contain multiple inputs (i.e., green boxes in Figure 1.1) and outputs (i.e., pink boxes in Figure 1.1), hundreds of blocks and lines, and are often hierarchical i.e., including subsystems (i.e., grey boxes in Figure 1.1). Furthermore, subsystems and their connecting lines may form a closed-loop structure. Complex structure of Simulink models makes debugging highly difficult and time-consuming. Hence, having a technique that is able to automatically locate faults in Simulink models is beneficial for engineers and reducing their debugging effort.

Statistical debugging is a light-weight and well-studied approach to fault localization in code([Abreu et al., 2007, Jones et al., 2002, Liblit et al., 2005, Liu et al., 2005, Renieris and Reiss, 2003, Santelices et al., 2009, Wong et al., 2014, Wong et al., 2008]). This approach utilizes an abstraction of program behavior, also known as *spectra* (e.g., sequences of executed statements), obtained from test case executions. The spectra and the testing results, in terms of failing or passing test cases, are used to derive a statistical fault ranking, specifying an ordered list of program elements (e.g., statements) likely to be faulty. Engineers (e.g., developers) consider such ranking to identify faults in their code. (see more information in Section 2.3)

In this dissertation we investigate and develop approaches based on statistical debugging, a light-weight fault localization technique, to automatically and accurately suggest the location of faults in Simulink models. We also implement a tool with informative and convenient user interfaces that can provide engineers with a better understanding about faults in their models.

1.3 Challenges

Debugging is a cumbersome and time-consuming task. There is a wide range of techniques in the literature for debugging and fault localization in source code [Ball et al., 2003, Cleve and Zeller, 2000, Groce et al., 2004, Hildebrandt and Zeller, 2000, Orso et al., 2003, Parnin and Orso, 2011, Renieris and Reiss, 2003, Zhang et al., 2006, Zhang et al., 2003]. However, none of these techniques have been previously applied to Simulink models. Currently, in most industrial companies, debugging Simulink models is limited to executing the Simulink models for a small number of test cases, and manually inspecting the results of each simulation by engineers. The executed test cases are often based on engineers' domain knowledge, experience, and intuition, but in a rather ad hoc way.

The goal of this dissertation is to develop techniques that help engineers identify faults in Simulink models with small debugging effort. In order to achieve this goal, we need to investigate and develop techniques that are able to automatically and accurately locate faults in Simulink models within reasonably short time duration. Specifically, we need to address the following challenges:

- The fault localization approaches we proposed in this dissertation are based on statistical debugging. Statistical debugging is most effective when it is provided with a large number of observation points (i.e., the spectra size). Existing approaches, where each test case produces one spectrum, require a large test suite to generate a large number of spectra. For Simulink models, however, test suites are typically small. This is mostly because test oracles for embedded software are costly, and further, test suites are required to be eventually applied at the Hardware-in-the-Loop stage where test execution is time-consuming and expensive. Hence, we may not obtain a sufficiently large number of spectra if we simply generate one spectrum per each test case as is the case in most existing work [Wong et al., 2014, Abreu et al., 2007, Jones et al., 2002, Liblit et al., 2005, Liu et al., 2005, Renieris and Reiss, 2003, Santelices et al., 2009].
- Statistical debugging often fails to properly deal with multiple faults because it implicitly assumes that all failures are caused by the same fault(s). However, in the presence of multiple faults, different failures might be due to different faults and faults might mask one another.
- One promising way to improve the accuracy of fault localization based on statistical debugging is to increase diversity among test cases in the underlying test suite. However, in Simulink, adding test cases is not a cost-free option because, in many situations, test oracles are developed manually or running test cases is expensive. In additionally, as noted in the literature [Campos et al., 2013], adding test cases does not always improve statistical debugging results. Given that in our context test oracles are expensive, can we find a way to stop test generation when adding new test cases is unlikely to bring about noticeable improvements in the fault localization results?

1.4 Research Contributions

In this dissertation, we addressed the challenges of fault localization for Simulink models. Specifically, we make the following contributions:

1. An approach to localize faults in Simulink models (assuming that they contain a single fault) based on statistical debugging and dynamic slicing. Our approach is the first to extend statistical

debugging to Simulink models. Specifically, we use dynamic slicing to generate Simulink model spectra such that each spectrum is related to one output and one test case, apply statistical ranking formulas to the resulting spectra to compute suspiciousness scores for each Simulink model block. We also propose iSimFL, an iterative fault localization approach to refine rankings by increasing the number of observed outputs at each iteration. Our approach utilizes a heuristic stopping criterion to avoid unnecessary expansion of test oracles. This contribution has been published in a journal paper [Liu et al., 2016b] and is discussed in Chapter 3.

2. A new technique to localize faults in Simulink models containing multiple faults using a supervised learning technique (decision trees learning). Our approach builds on statistical debugging and is iterative. At each iteration, we identify and resolve one fault and re-test models to focus on localizing faults that might have been masked before. We use decision trees to cluster together failures that satisfy similar (logical) conditions on model blocks or inputs. We then present two alternative selection criteria to choose a cluster that is more likely to yield the best fault localization results among the clusters produced by our decision trees. This contribution has been published in a conference paper [Liu et al., 2016a] and is presented in Chapter 4.
3. A search-based testing technique for Simulink models that uses three existing alternative test objectives to generate small and diverse test suites that can help improve fault localization accuracy. We also develop a strategy to stop test generation when test generation is unlikely to improve fault localization. Our strategy builds on static analysis of Simulink models and prediction models are based on supervised learning. This contribution has been published in a conference paper [Liu et al., 2017], and is presented in Chapter 5. We have also submitted an extension version of our paper to Empirical Software Engineering journal.
4. Simulink Fault Localization Tool (*SimFL*) is the tool that implements our fault localization approaches and test generation approaches for Simulink models. The description of *SimFL* is included in our journal paper submission and is presented in Chapter 6.
5. We have evaluated all of our proposed techniques by applying them into real industrial case studies from our industrial partner, Delphi Automotive Systems Luxembourg.

1.5 Concepts and Definitions

Throughout this dissertation, we use the following terminology.

1. A *failure* is the situation where a model output deviates from the expected/correct output.
2. A *fault (defect/bug)* is the cause of a failure in the model.
3. A *test oracle* is a predicate that determines whether a given test activity is an acceptable behavior of the SUT (System Under Test) or not [Barr et al., 2015].
4. A *test suite* is a set of test cases.

The prerequisite for triggering a fault localization process is that a failure is observed/occurred for a model. In practice, although a software may not be fault free, it is un-reasonable to trigger a fault localization technique without any failure observed because many bugs in the software remain latent or never lead to a failure [Abreu, 2009].

Note that, in practice, the development of test oracles is effort intensive and out of scope of this dissertation. In our experiments, we assume test oracle information is available, and chose to use a fault-free version of our industrial subject model for the oracle information to automate our large-scale and time-consuming experiments.

1.6 Organization of the Dissertation

Chapter 2 provides some background information on modeling of Cyber-physical systems, Matlab/Simulink model, Statistical debugging, Decision tree learning techniques, and Single-state meta-heuristic search algorithms.

Chapter 3 describes our approach to localize faults in single-fault Simulink models based on statistical debugging.

Chapter 4 describes our approach for localizing multiple faults in Simulink models.

Chapter 5 describes our test case generation algorithms for developing small and effective test suites for improving fault localization of Simulink models.

Chapter 6 presents the tool we developed which implements the approaches we proposed to localize faults in Simulink models.

Chapter 7 presents and discusses related work.

Chapter 8 summarizes the thesis contributions and discusses perspectives on future work.

Chapter 2

Background

In this chapter, we present some background information that are necessary to understand the techniques/approach and key concerns that we have conducted in this dissertation. The content of this chapter is organized under following headings: (1) Cyber-physical System Modeling and Simulation, (2) Matlab/Simulink, (3) Debugging and Fault Localization, (4) Supervised Learning techniques, (5) Single-state meta-heuristic search algorithm.

2.1 Cyber-physical System Modelling and Simulation

A Cyber-Physical System is a system that integrates physical and cyber components to realize relevant functions through the interactions between the physical and cyber parts [Lee, 2008]. Nowadays, Cyber-Physical Systems (CPSs) exist in different aspects of our lives. From smart household appliances to advanced satellite systems, these CPSs are becoming more and more complicated and costly. This trend also brings some new challenges for engineers to design and test CPSs: testing a highly-complicated and safety-critical system with actual hardware can be very costly, time-consuming, and even risky. Take nuclear power plant systems as examples, building such systems is risky for engineers' lives especially if they don't have enough confidence in the correctness of their design.

As a result, in practice, designing and testing CPSs usually start from system modeling. System modeling is a process to abstract and build a mathematical description of a real system. These models are built to mimic the real behaviors of systems under different operating conditions [Chaturvedi,

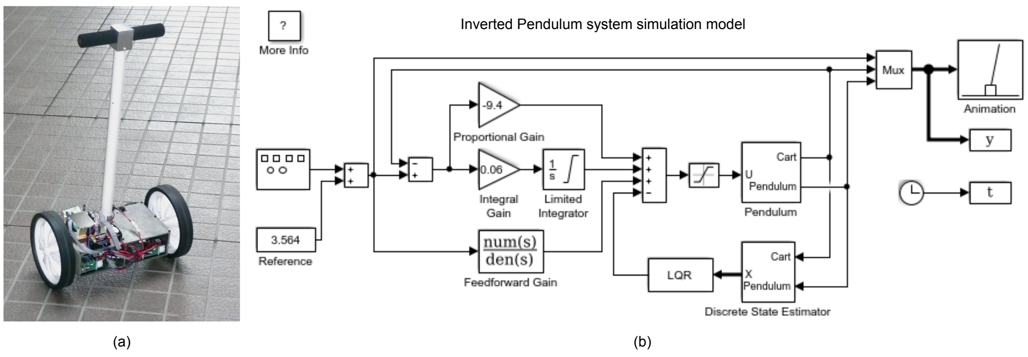


Figure 2.1. An real inverted pendulum system application (a) example and related simulation model in Simulink(b).

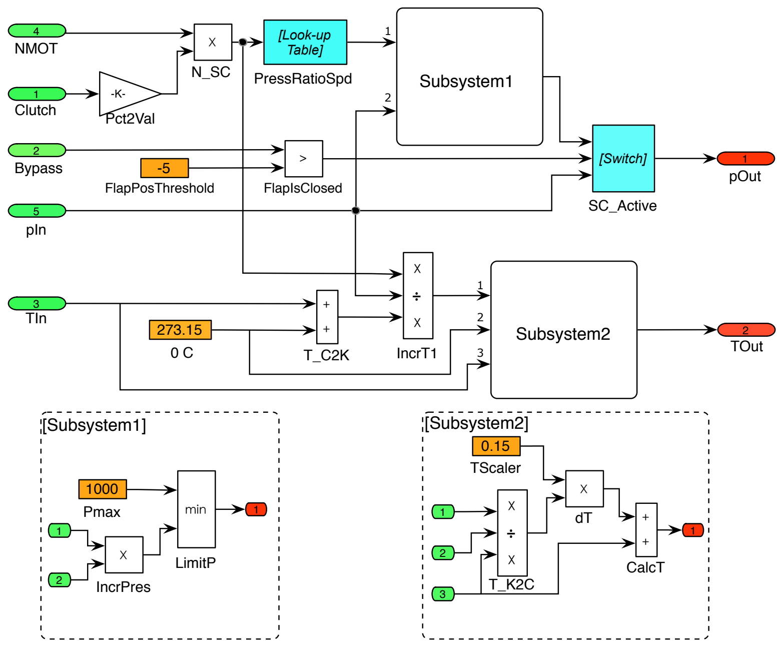


Figure 2.2. A Simulink model example.

2009]. Moreover, once engineers build a model of a real system, they can conduct simulations by feeding the model with different inputs to observe and capture the behavior of the system. Model simulation is particularly useful to enable design-time testing and detect faults when a real system has not been implemented. For example, the application shown in Figure 2.1(a) is a typical usage scenario of inverted pendulum system, and engineers could first build a system model as shown in Figure 2.1(b) to check and tune the coefficients of their Proportional-Integral (PI) controller implementation.

In order to obtain an accurate system model, the model usually has to go through several rounds of checking and refinement. It is crucial to offer the engineers with proper tools and techniques to test and debug their models. In this thesis, we provide effective techniques to help engineers effectively find out the root cause of failures in their model after any abnormal behavior has been observed in the simulation process.

2.2 Matlab/Simulink

MATLAB/Simulink is an advanced environment for modeling, simulating and analyzing multidomain dynamic systems [Mathworks, 2015], it has been widely used to model the advanced Cyber-Physical Systems, e.g., in the automotive industry or avionics industry. Particularly, Simulink, included in each Matlab release, provides both a graphical programming interface as well as a customizable set of block libraries to model CPSs. Figure 2.2 shows an example of a Simulink model. As shown in the example, Simulink models consist of blocks and lines. Blocks may perform individual operations such as numerical and combinatoric operations or they may represent constant values e.g., the *Pmax* block. Simulink blocks are connected via lines that indicate data flow connections. The model has five inputs, e.g., the intake air pressure *pIn*, and two outputs: the output pressure *pOut* and the output temperature *TOut*. Moreover, Simulink models can be hierarchical and allow the encapsulation of blocks into subsystems (e.g., *Subsystem1* in Figure 2.2). Each subsystem has its own input and output

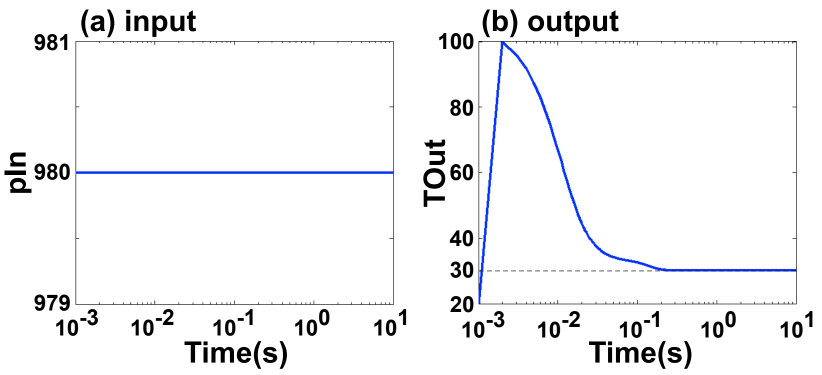


Figure 2.3. Example of an input (a) and an output (b).

ports.

2.2.1 Simulink Input data

Engineers simulate (execute) the Simulink model by providing input signals, i.e., functions over time. In theory, the input signals can be complex continuous functions. In practice, however, engineers mostly test Simulink models using constant input signals over a fixed time interval. This enables engineers to reproduce the simulation results on different platforms (e.g., when the environment is composed of real hardware or is a real-time simulator). Further, developing test oracles for non-constant input signals is very complex and time-consuming. Figure 2.3(a) shows an input signal example applied to the input *pIn*. The input signal time interval indicates the simulation length and is chosen to be large enough to let the output signals stabilize.

2.2.2 Simulink Test output

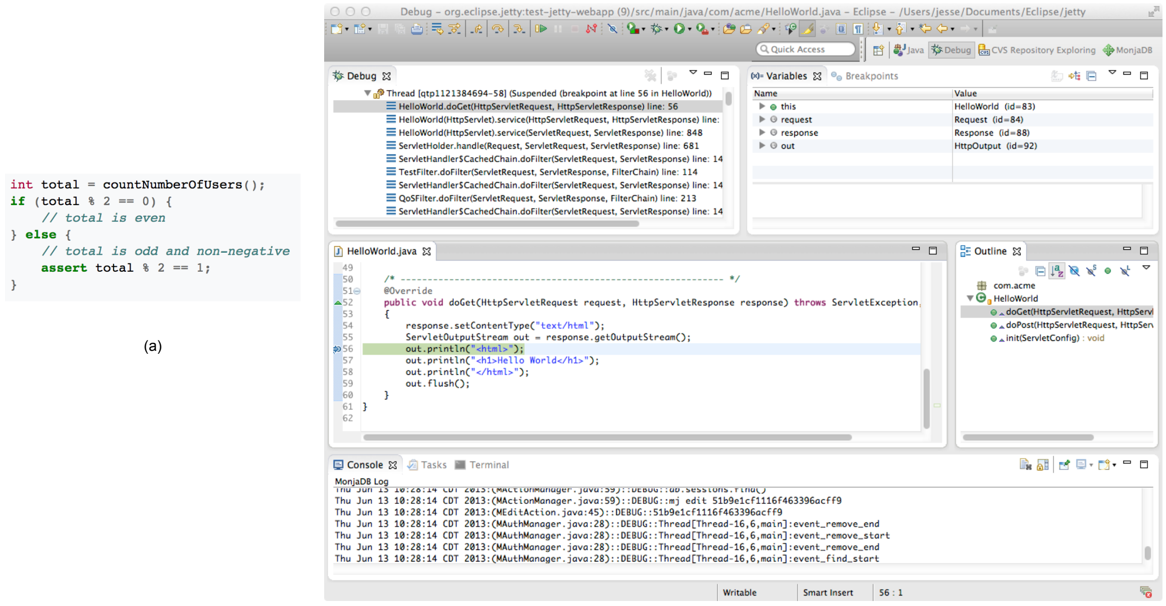
Similar to the input, the Simulink model output is a signal. Each test case execution (simulation) of a Simulink model results in an individual output signal for each output port of that model. Engineers evaluate each output signal independently. To determine whether an output passes or fails a test case, engineers evaluate various aspects of the output signal, particularly the value at which the output signal stabilizes (if it stabilizes) and the dynamic characteristics of the signal, such as the signal fluctuations (over/undershoot), the response time, and if the signal reaches a steady state. For example, Figure 2.3(b) shows an example output signal of *TOut*. As shown in the figure, the output signal stabilizes after 1 sec of simulation. The output values are the final (stabilized) values of each output signal collected at the end of simulation (e.g., 30 for the signal shown in Figure 2.3(b)).

2.3 Debugging and Fault Localization

Debugging and Fault Localization is a process to identify and resolve the fault(s)/defect(s) that lead to failures or abnormal behavior in a software system [Wikipedia, 2017a]. When a failure is observed, developers have to conduct a debugging task to identify the root cause of the failure. This kind of activities has performed since the first day of computer history.

In general, debugging is a cumbersome and time-consuming task. The effectiveness of debugging tasks depends on several factors, such as the developers' understanding of the artifact (model or program) being debugged, the developers' personal debugging experiences, the quality of the test suite, and etc. [Wong et al., 2016]

A traditional and intuitive fault localization technique is to log or instrument the program under test to collect additional execution information to help developers identify the cause of failures [Abreu, 2009, Wong et al., 2016]. For example, developers could insert print statement or assertions manually in the program to return variable values or check whether program reaches any specific runtime state as shown in Figure 2.4 (a), or use the debug prospect/view in the IDE(as shown in Figure 2.4 (b)). However, these traditional debugging activities are manual, hence they heavily rely on developers' intuition and need a lot of human effort. In order to avoid these limitations and make the debugging approach applicable to realistic programs, many advanced fault localization techniques have been proposed and studied specifically for software code [Ball et al., 2003, Liblit et al., 2005, Abreu et al., 2007, Abreu et al., 2009b, Alves et al., 2011].



```
int total = countNumberOfUsers();
if (total % 2 == 0) {
    // total is even
} else {
    // total is odd and non-negative
    assert total % 2 == 1;
}
```

(a)

Debug – org.eclipse.jetty:test-jetty-webapp (9)/src/main/java/com/acme/HelloWorld.java – Eclipse – /Users/jesse/Documents/Eclipse/jetty

```
Thread [qtp1121384694-58] (Suspended (breakpoint at line 56 in HelloWorld))
    HelloWorld.doGet(HttpServletRequest, HttpServletResponse) line: 56
    HelloWorld(HttpServlet).service(HttpServletRequest, HttpServletResponse) line:
    HelloWorld(HttpServlet).service(ServletRequest, ServletResponse) line: 848
    ServletHolder.handle(Request, ServletRequest, ServletResponse) line: 681
    ServletHandler$CachedChain.doFilter(ServletRequest, ServletResponse) line: 14
    TestFilter.doFilter(ServletRequest, ServletResponse, FilterChain) line: 114
    ServletHandler$CachedChain.doFilter(ServletRequest, ServletResponse) line: 14
    QoSFilter.doFilter(ServletRequest, ServletResponse, FilterChain) line: 213
    ServletHandler$CachedChain.doFilter(ServletRequest, ServletResponse) line: 14
```

Name	Value
this	HelloWorld (id=83)
request	Request (id=84)
response	Response (id=88)
out	HttpOutput (id=92)

HelloWorld.java

```
    /* ------------------------------------------------------------ */
    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException,
    {
        response.setContentType("text/html");
        ServletOutputStream out = response.getOutputStream();
        out.println("<html>");
        out.println("<h1>Hello World</h1>");
        out.println("</html>");
        out.flush();
    }
}
```

```
com.acme
  HelloWorld
    doGet(HttpServletRequest, HttpServl
    doPost(HttpServletRequest, HttpServ
    init(ServletConfig) : void
```

MonjaDB Log

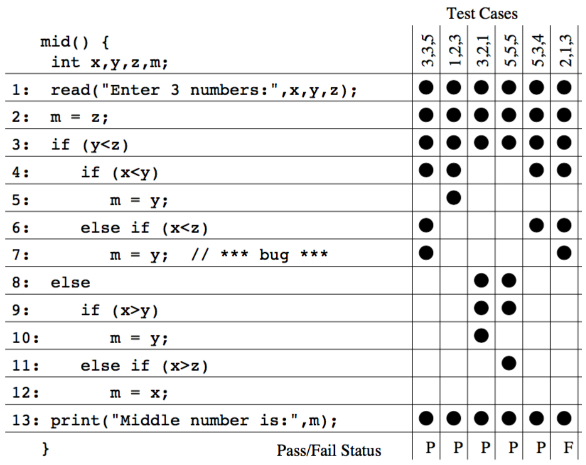
```
Thu Jun 13 10:28:14 CDT 2013:(MActionManager.java:59)::DEBUG::db.sessions.find()
Thu Jun 13 10:28:14 CDT 2013:(MActionManager.java:59)::DEBUG::mj edit 51b9e1cf1116f463396acff9
Thu Jun 13 10:28:14 CDT 2013:(MEditAction.java:45)::DEBUG::51b9e1cf1116f463396acff9
Thu Jun 13 10:28:14 CDT 2013:(MAuthManager.java:28)::DEBUG::Thread[Thread-16,6,main]:event_remove_end
Thu Jun 13 10:28:14 CDT 2013:(MAuthManager.java:28)::DEBUG::Thread[Thread-16,6,main]:event_remove_start
Thu Jun 13 10:28:14 CDT 2013:(MAuthManager.java:28)::DEBUG::Thread[Thread-16,6,main]:event_remove_end
Thu Jun 13 10:28:14 CDT 2013:(MAuthManager.java:28)::DEBUG::Thread[Thread-16,6,main]:event_find_start
```

(b)

Figure 2.4. Examples of inserted assertion (a) and debug view in IDE Eclipse (b).

Our fault localization approaches proposed in this thesis are based on statistical debugging techniques. Hence, in this section, we focus on explaining the statistical debugging techniques. Other fault localization techniques will be discussed in Chapter 7.

Statistical debugging is a light-weight approach to fault localization and has been extensively studied for code (e.g., C programs [Abreu et al., 2007, Jones et al., 2002, Renieris and Reiss, 2003, Zoeteweij et al., 2007]). This approach utilizes an abstraction of program behavior, also known as spectra, (e.g., sequences of executed statements) obtained from testing. The spectra and the testing results, in terms of failed or passed test cases, are used to derive a statistical fault ranking, specifying an ordered list of program elements (e.g., statements) likely to be faulty. Developers can consider such rankings to identify faults in their code.



	`mid() {` `int x,y,z,m;`	Test Cases 3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3
1:	`read("Enter 3 numbers:",x,y,z);`	●	●	●	●	●	●
2:	`m = z;`	●	●	●	●	●	●
3:	`if (y<z)`	●	●	●	●	●	●
4:	`if (x<y)`	●	●			●	●
5:	`m = y;`		●				
6:	`else if (x<z)`	●				●	●
7:	`m = y;  // *** bug ***`	●					●
8:	`else`			●	●		
9:	`if (x>y)`			●	●		
10:	`m = y;`			●			
11:	`else if (x>z)`				●		
12:	`m = x;`						
13:	`print("Middle number is:",m);`	●	●	●	●	●	●
	`}` Pass/Fail Status	P	P	P	P	P	F

Figure 2.5. Example of statistical debugging for program code.

In order to explain how statistical debugging works, we use a simple example program, which has been originally used in the work of [Jones et al., 2002]. In Figure 2.5, the program *mid()* takes three integers as inputs and return the median value. The faulty statement is *line*7 (the line should read " *m = x;* ". On the right side of this Figure, we show the spectra information of each test case in the underlying test suite. The corresponding input values are listed on the top of each column, the detailed spectrum of each test case are marked with *black solid dots*, and the pass/fail information was shown at the bottom of each column.

Obviously, compared to *line*4, *line*8 is less suspicious because no failed test case execution exercises *line*8. Similarly, *line*4 is less suspicious compared to *line*7 because more passed test case executions exercise *line*4 than *line*7. So the probability that leads to a failure is higher for *line*7 than other statements in this program. *line*7 is the first statement recommended by statistical debugging technique to engineers to inspect.

Although statistical debugging techniques have been extensively studied to show their effectiveness in debugging program code, these techniques have never been studied for Simulink models. In this thesis, we propose approaches to localize faults in Simulink models based on statistical debugging in both single-fault and multiple-fault situations.

2.4 Supervised Learning techniques

Machine learning is a type of artificial intelligence (AI) and is the science that provides computers with the ability to learn from the existing data and experience without being explicitly programmed [Wikipedia, 2017b]. The fundamental goal of machine learning is to generalize beyond the examples in the training set [Domingos, 2012]. In machine learning, based on the information included in the training data set, machine learning tasks can be classified into several categories, and supervised learning is one type of machine learning tasks.

When the training data includes a set of examples with paired input subjects and desired output, we call it *supervised learning*. Supervised learning is the task of inferring a mapping (function) between a set of input objects and output variables from labeled training data [Alpaydin, 2014]. These mapping relations will be later applied to predict the outputs for unseen (new) data/objects.

In this dissertation, we use a supervised learning technique, namely decision trees, for two objectives: (1). Clustering the failed test execution slices (in Chapter 4); (2). Building prediction models (decision trees) based on historical data (in Chapter 5).

A decision tree is a hierarchical tree structure implementing the divide-and-conquer strategy on the input objects [Alpaydin, 2014]. In another word, a decision tree is a predictor, $h : X \rightarrow Y$, that predicts the label (decision) associated with an instance x by traveling from a root node of a tree to a leaf [Shalev-Shwartz and Ben-David, 2014]. Decision trees can be used for both classification and regression. Decision trees are composed of leaf nodes, which represent partitions, and non-leaf nodes, which represent decision variables. Usually, the splitting is based on one of the features of x (marked in the non-leaf nodes). Each leaf node contains a specific label indicating the decision (or prediction result).

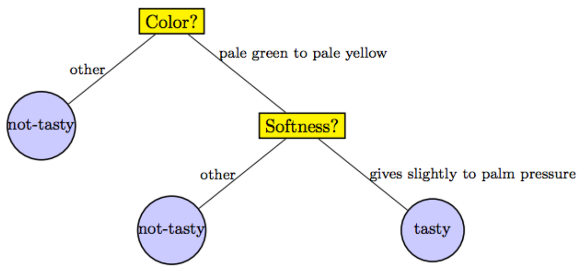


Figure 2.6. An example of decision tree for distinguishing papayas.

Figure 2.6 shows an example of a decision tree for checking if a given papaya is tasty or not (this is an illustration example which was originally used in [Shalev-Shwartz and Ben-David, 2014]). In order to predict whether a given papaya is tasty or not, the color of the papaya will be first checked. If the color is not in the range from pale green to pale yellow, the tree will immediately predict that the papaya is not tasty. Otherwise, the second step is to check the softness of the papaya. The decision tree predicts the papaya is tasty when the papaya gives slightly to palm pressure. Otherwise, the prediction result is "not-tasty". A decision (label) is made by following one of the paths from the root to a leaf where all the condition in this path can be completely satisfied by the corresponding features of an input.

2.5 Single-state meta-heuristic search algorithm

Meta-heuristic search is a procedure or heuristic to search or find optimal (or as optimal as possible) solutions to hard problems [Luke, 2015]. It has been applied to a very wide range of optimization problems, where no prior information and guideline about what the optimal solution looks like and how to approach an optimal solution are available, or where brute-force search is infeasible because the input search space is too large. In order to obtain an optimal solution, meta-heuristic search algorithms, in general, can be explained in four steps [Luke, 2015]:

- 1. *Initialization procedure*: creating one or more initial candidate solutions;

- 2. *Modification procedure*: tweaking a candidate solution, which produces a randomly slightly different candidate solution;
- 3. *Assessment procedure*: assessing the quality of each candidate solution (computing fitness function);
- 4. *Selection procedure*: deciding which candidate solution to retain;

The search process is iterative, steps 2 - 4 will be iterated until any of the pre-defined stop conditions is satisfied (e.g., timeout) or an optimal solution is found.

Meta-heuristics algorithms can be classified into *Single-state search* and *Population-based search*. *Single-state search* keeps only one candidate solution in each iteration. In contrast, *Population-based search* keeps a sample (or a set) of candidate solutions rather than a single candidate in each iteration. In our work, computing fitness function (Step 3) requires us to execute the simulation on a given Simulink model. This is computationally expensive and takes a long time. Hence, we rely on single-state search as opposed to population-based search.

In the rest part of this subsection, we describe two single-state meta-heuristic search algorithms that we used in our work: Hill-Climbing (HC), and Hill-Climbing with Random Restarts (HCRR).

2.5.1 Hill-Climbing (HC)

Figure 2.7 shows the Hill-Climbing (HC) algorithm. Initially, we randomly generate an initial candidate solution S as the temporary best solution (The Initialization Procedure). Then, in each iteration, a new candidate R is generated by slightly modifying the current best solution S as shown at $line3$ (The Modification Procedure). In the following assessment procedure, the temporary best solution S is replaced by the new candidate solution R only if the quality of R is better than the quality of S, i.e., R fits better to the optimization goal. The termination conditions of this algorithm are: either (1) S is the ideal solution, or (2) the pre-defined budget has been reached.

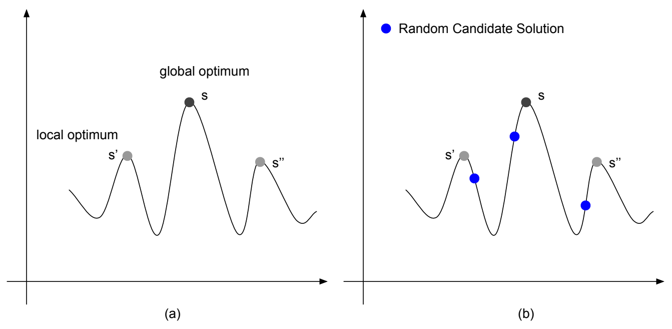
```
Algorithm  Hill-Climbing
1: $S \leftarrow$ some initial candidate solution                ▷ The Initialization Procedure
2: repeat
3:     $R \leftarrow$ Tweak(Copy($S$))                             ▷ The Modification Procedure
4:     if Quality($R$) > Quality($S$) then          ▷ The Assessment and Selection Procedures
5:         $S \leftarrow R$
6: until $S$ is the ideal solution or we have run out of time
7: return $S$
```

Figure 2.7. Hill-Climbing (HC) procedure.

Hill-Climbing algorithm enables the search to climb up the hill and effectively reach a local optimum. However, for a more complex situation (as shown in Figure 2.8(a)) to find out a global optimal solution, Hill-Climbing algorithm might not be efficient enough.

2.5.2 Hill-Climbing With Random Restarts (HCRR)

As discussed in the previous section, the Hill-Climbing algorithm may stick at a local optimum (e.g., the s' or s'' in Figure2.8(a)) since the *Tweak* operation conceptually is to "make a small, bounded, but random change" [Luke, 2015]. When local optimum is sufficiently broad, *Tweak* modification



```
Algorithm    Hill-Climbing with Random Restarts
 1: $T \leftarrow$ distribution of possible time intervals

 2: $S \leftarrow$ some initial random candidate solution
 3: $Best \leftarrow S$
 4: repeat
 5:     $time \leftarrow$ random time in the near future, chosen from $T$
 6:     repeat
 7:         $R \leftarrow$ Tweak(Copy($S$))
 8:         if Quality($R$) > Quality($S$) then
 9:             $S \leftarrow R$
10:     until $S$ is the ideal solution, or $time$ is up, or we have run out of total time
11:     if Quality($S$) > Quality($Best$) then
12:         $Best \leftarrow S$
13:     $S \leftarrow$ some random candidate solution
14: until $Best$ is the ideal solution or we have run out of total time
15: return $Best$
```

(c)

Figure 2.8. Hill-Climbing with Random Restarts (HCRR) examples and procedure.

may not be large enough to get the search out of a plateau in the fitness function. The root reason of this limitation in Hill-Climbing algorithm is that HC is extreme in exploitation but lacks exploration. To avoid these limitations in Hill-Climbing algorithm, another algorithm, called Hill-Climbing with Random Restarts algorithm (HCRR), is proposed.

In essence, HCRR is a mixture of exploitation and exploration. HCRR consists of a series of Hill-Climbing searches from different random initial positions. Specifically, in each HC search process of HCRR algorithm, a local optimal candidate solution is voted out (as described in Figure 2.8(c) $line6$ – $line10$); then these local optimal solutions are compared and finally a global optimal solution is identified (as described in Figure 2.8(c) $line11$ – $line13$). Take Figure 2.8(b) as an example, assume that in three HC search iterations, s', s'', and s are selected as each local optimal candidate solution. Among these three candidates solutions, s has the highest fitness and will be identified as the global optimal solution.

Chapter 3

An Iterative Statistical Debugging Approach for Simulink Fault localization

In this chapter, we propose, SimFL, a combination of statistical debugging and dynamic slicing to localize faults in Simulink models. Our work is the first to extend statistical debugging to Simulink models. We use dynamic slicing to generate Simulink model spectra such that each spectrum is related to one output and one test case. We then apply statistical ranking formulas to the resulting spectra to compute suspiciousness scores for each Simulink model block. Moreover, we propose, *iSimFL*, an iterative fault localization approach to refine rankings by increasing the number of observed outputs at each iteration. Our approach utilizes a heuristic stopping criterion to avoid unnecessary expansion of test oracles.

We have implemented our approach and evaluated our approach by applying our approach to three *industrial* subjects. This is the first empirical study to evaluate statistical debugging for Simulink models using *industrial* case studies. Our experiments show that our technique is able to accurately locate faults in Simulink models, particularly in single-fault Simulink models. Further, our technique is light-weight and hence scalable to large Simulink models.

This chapter highlights the following research contributions:

1. We propose *SimFL*, a combination of statistical debugging and dynamic slicing to localize faults in Simulink models.
2. We propose *iSimFL*, an iterative fault localization approach to refine rankings by increasing the number of observed outputs at each iteration. Our approach utilizes a heuristic stopping criterion to avoid unnecessary expansion of test oracles.
3. We conduct, for the first time, an empirical study to evaluate statistical debugging for Simulink models using three industrial subjects.

Organization. This chapter is organized as follows. Section 3.1 precisely formulates the problem we aim to address in this chapter. Section 3.2 presents our approach to fault localization in Simulink models. Section 3.3 describes an iterative fault localization approach, namely *iSimFL* that can further improve the accuracy of *SimFL* in localizing faults. The results of our evaluation of the proposed approaches are presented in Section 3.4. Finally, Section 3.6 concludes the chapter.

3.1 Problem Formulation

Fault localization in source code has been an active research area that focuses on automating various code debugging activities [Abreu et al., 2007, Ball et al., 2003, Cleve and Zeller, 2000, Cleve and Zeller, 2000, Groce et al., 2004, Hildebrandt and Zeller, 2000, Jones et al., 2002, Liblit et al., 2005, Liu et al., 2005, Orso et al., 2003, Parnin and Orso, 2011, Renieris and Reiss, 2003, Santelices et al., 2009, Wong et al., 2008, Wong et al., 2014, Zhang et al., 2006, Zhang et al., 2003]. A well-known approach in this area is statistical debugging [Abreu et al., 2007, Jones et al., 2002, Liblit et al., 2005, Liu et al., 2005, Renieris and Reiss, 2003, Santelices et al., 2009, Wong et al., 2008, Wong et al., 2014]. Statistical debugging is a lightweight approach to fault localization and has been extensively studied for code (e.g., C programs [Abreu et al., 2007, Jones et al., 2002, Renieris and Reiss, 2003, Zoeteweij et al., 2007]). This approach utilizes an abstraction of program behavior, also known as spectra, (e.g., sequences of executed statements) obtained from testing. The program spectra as well as the testing results, in terms of failed or passed test cases, are used to derive a statistical fault ranking, specifying an ordered list of program elements (statements, blocks, etc.) likely to be faulty. Developers can consider such ranking to identify faults in their code. These fault localization techniques, however, have never been studied for Simulink models.

Statistical debugging is most effective when it is provided with a large number of observation points (i.e., the spectra size). The existing techniques, where each test case produces one spectrum, require a large test suite to generate a large number of spectra. For Simulink models, however, test suites are considerably smaller than test suites used for generic or open source software. This is mostly because test oracles for embedded software are costly, and further, test suites are required to be eventually applied at the Hardware-in-the-Loop stage where test execution is time consuming and expensive. Hence, we may not obtain a sufficiently large number of spectra if we simply generate one spectrum per each test case.

Simulink models, being visual, dataflow based and hierarchical, have multiple observable outputs at different hierarchy levels, each of which can be tested and evaluated independently. For each given test case, engineers routinely and explicitly determine which specific outputs are correct and which ones are incorrect. Relying on this observation, in our work, we use a dynamic slicing technique in conjunction with statistical debugging to generate one spectrum per each output and each test case. Hence, we obtain a set of spectra that is significantly larger than the size of the test suite. We then use this set of spectra to rank blocks using statistical ranking formulas.

3.2 Description of the Approach

We present *SimFL*, our fault localization approach for Simulink models. Figure 3.2 shows an overview of *SimFL*. The inputs to our approach is a (faulty) Simulink model (M), a test suite ($TS = \{tc_0, \dots, tc_n\}$), and a test oracle ($\mathcal{O}$) to determine whether the test cases in TS pass or fail.

Given a Simulink model M, we denote the set of input ports of M by I. For the model in Figure 3.1, the set I is {`NMOT, Clutch, Bypass, pIn, TIn`}, and each test case in TS provides a value (i.e., a constant signal) for each element in I. We denote the set of all outputs of M by O, including the model outputs (at depth zero) as well as all the subsystem outputs. For each test case $tc \in TS$, the test oracle $\mathcal{O}$ determines whether each output $o \in O$ passes or fails tc.

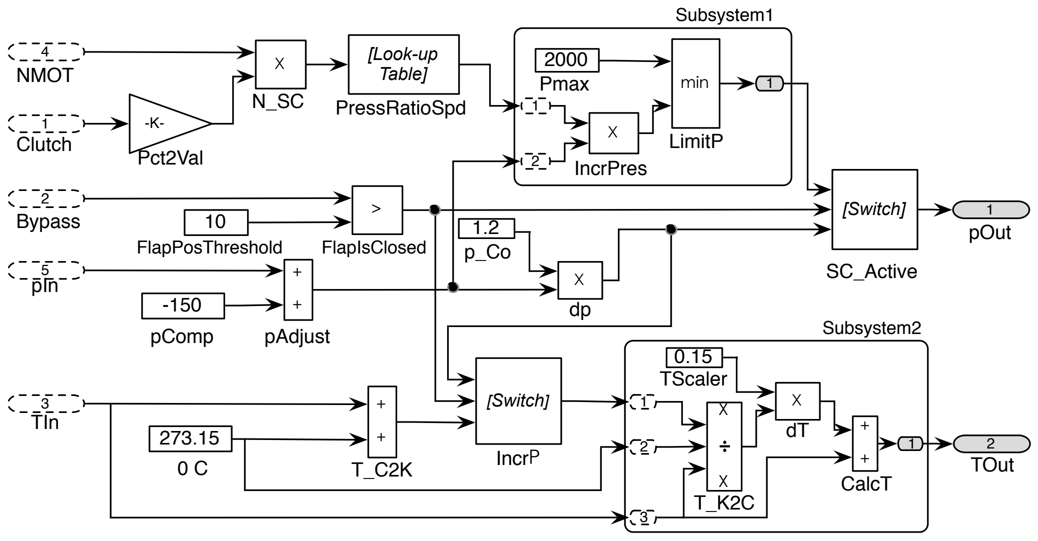


Figure 3.1. A snippet of a real-world Simulink model.

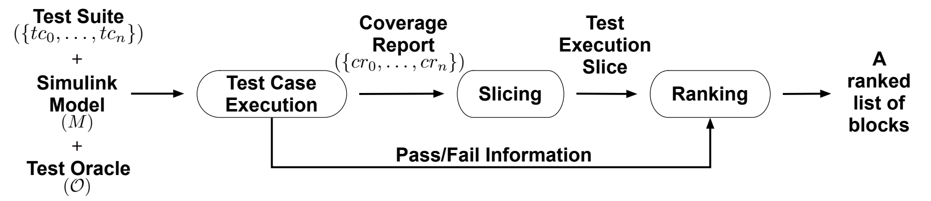


Figure 3.2. Overview of our fault localization approach for Simulink (*SimFL*).

The output of the approach in Figure 3.2 is a ranked list of Simulink (atomic) blocks where the top ranked blocks are more likely to be faulty. This ranked list is generated based on the three main steps of *SimFL*, i.e., Test Case Execution, Slicing, and Ranking, that we discuss in Sections 3.2.1 to 3.2.3, respectively.

3.2.1 Test Case Execution

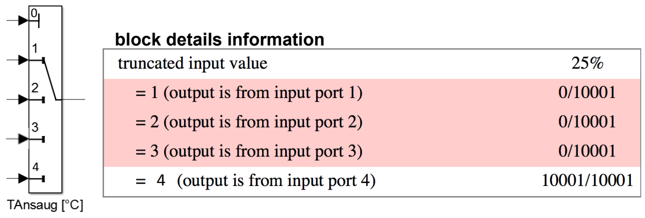
This step takes as input a test suite *TS*, a test oracle $\mathcal{O}$, and a (faulty) Simulink model *M*. In this step, we execute *M* for each test case in *TS* to generate the following information: (1) The PASS/FAIL information corresponding to each output *o* of *M* and each test case in *TS*, and (2) A list $\{cr_0, \ldots, cr_n\}$ of *coverage reports* corresponding to the test cases $\{tc_0, \ldots, tc_n\}$.

In Section 2.2.2, we discussed how Simulink output signals are typically evaluated to obtain the PASS/FAIL information. In this section, we focus on coverage reports. Given a test case tc_l, Simulink generates a coverage report cr_l after simulating *M* using tc_l. A coverage report shows the list of atomic blocks that were covered during execution of tc_l.

Using a coverage report describing a list of atomic blocks covered by a test case, we identify which inputs of those blocks were covered by that test case as well. Simulink atomic blocks have two kinds of inputs: data inputs and control inputs. Every (non trivial) atomic block has some data inputs[1]. But they may or may not have control inputs. For a block that has only data inputs, e.g., a multiplication, we know that all its inputs are covered if that block is covered, i.e., appears in the coverage report.

[1]Some trivial Simulink blocks (e.g., clock) do not have any input.

For a block that has control inputs as well as data inputs, e.g., a switch block, the coverage report provides some *block details information* describing which data inputs were covered and which ones were not covered. For example, Figure 3.3 shows the block details information for a MultiPortSwitch block, which consists of five inputs: one control input (i.e., input 0) and four data inputs (i.e., inputs 1–4). The table in Figure 3.3 reports which data inputs were actually covered during simulation. Specifically, inputs 1–3 (highlighted in red by Simulink) were not covered, whereas input 4 was covered. That is, the control input 0 selected the data input 4 for the output. Note that the coverage report does not explicitly include the control input 0. However, we know that all the control inputs of a covered block are covered as well. To summarize, from the coverage report in Figure 3.3, we conclude that among inputs 0–4, only control input 0 and data input 4 were covered during simulation.



block details information

truncated input value	25%
= 1 (output is from input port 1)	0/10001
= 2 (output is from input port 2)	0/10001
= 3 (output is from input port 3)	0/10001
= 4 (output is from input port 4)	10001/10001

Figure 3.3. A coverage report snippet generated by Simulink

Note that coverage reports combine the list of covered blocks for all the Simulink outputs. That is, they do not determine which blocks were covered for which specific output. We use slicing (Section 3.2.2) to determine which blocks were covered for which output.

3.2.2 Slicing

The second step of *SimFL* is slicing of the input model. This step takes as input the Simulink model M, and the set of coverage reports $\{cr_0, \dots, cr_n\}$ from the first step. The output of this step is a set of test execution slices of M indicating the set of (atomic) blocks that were executed by each test case to generate each output in O.

To generate test execution slices, we first generate a *backward static slice*, denoted by $static\_slice_o$, for each output $o \in O$. That is, we set the *slicing criterion*, which also indicates the starting point of the slice, to an output port. Important considerations in slicing are data and control dependencies [Reicherdt and Glesner, 2012]. As Simulink is a data-flow oriented language, data dependencies are specified by the links between blocks. Starting from an output port, we follow the data dependencies of blocks backwards through the model. If a block is data dependent, then we add this block to our slice. Note that in Simulink, the data dependency links are disconnected at subsystem input/output ports and at Goto/From blocks. In our backward graph traversal, for each subsystem, we ensure to connect its input (respectively, output) ports to the corresponding incoming (respectively, outgoing) links of that subsystem. Similarly, we connect the Goto ports to their matching From ports. The backward traversal stops once we reach the model input ports or constant blocks (shown as orange blocks in Figure 3.1). Finally, we note that Simulink models may contain feedback loops enabling to use the output of a block for a subsequent calculation [Reicherdt and Glesner, 2012]. The graphical structure of a Simulink model with a feedback loop is cyclic. To generate (backward) static slices, we detect these cycles and do not go through them more than once.

For example, the model in Figure 3.4 is an example of a static slice for a MultiPortSwitch block. Suppose that the slicing criterion is the output port `Taus`. To compute the static slice, we follow the (backward) data dependency to `TAnsaug`, i.e., the link between `TAnsaug` and `Taus`. Hence, we add `TAnsaug` to the slice. Next, we identify the data dependencies to the blocks `TAnsaugdaempfer`, `Ground`, and `pEin`, and thus, we add these three blocks to the slice. Note that blocks `TAnsaugdaempfer` and `Ground` are constant, and hence, do not induce any further data dependencies, while for the block `pEin`, we may have further data dependent elements that are not shown in the figure.

The backward static slices discussed above are generated for each individual output, but they may contain blocks that do not always affect that output at runtime. For example, the static slice in Figure 3.4 includes all the blocks connected to the MultiPortSwitch `TAnsaug`, however, for a given test case, only *some* of the blocks connected to this switch may be executed.

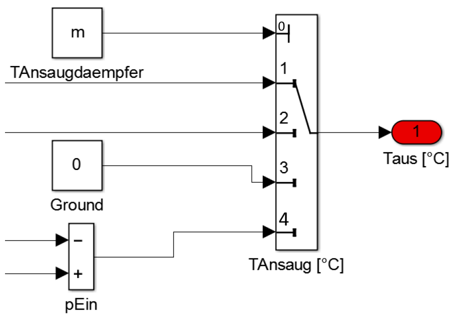


Figure 3.4. A (partial) static slice for a MultiPortSwitch block.

Having generated backward static slices for each output, we create test execution slices by identifying the subset of these static slices that are executed by each test case. Given an output $o \in O$ and a test case $tc_l \in TS$, in this step, we compute a test execution slice, denoted by $slice_{o,l}$, containing the blocks that are executed by tc_l to generate o. Let $\{cr_0, \ldots, cr_n\}$ be the set of coverage reports, and let $\{static\_slice_o \mid o \in O\}$ be the set of static backward slices. We define $slice_{o,l}$ as a set of atomics b as follows:

$$slice_{o,l} = \{b \mid b \in static\_slice_o \wedge b \in cr_l\}$$

We compute a test execution slice $slice_{o,l}$ by traversing the blocks in the backward static slice of o (i.e., $static\_slice_o$) and including in $slice_{o,l}$ those blocks that appear in the coverage report of tc_l (i.e., cr_l). We use cr_l to determine the behavior of Simulink blocks for each test case and for each output. For each block in the static slice, cr_l helps identify which data inputs of that block are selected by its control inputs. For example, as discussed in Section 3.2.1, the coverage report in Figure 3.3 indicates that among inputs 0 – 4, only the inputs 0 and 4 are covered during running a test case. Combining this coverage report with the static slice in Figure 3.4, we obtain a test execution slice which includes `TAnsaugdaempfer` and `pEin` blocks. That is, the block `Ground` is not included in the test execution slice.

A test execution slice $slice_{o,l}$ is passing (respectively failing) if the result of tc_l for output o matches (respectively deviates from) the test oracle for o. Table 3.1 shows eight test execution slices corresponding to four test cases (TC1 to TC4) and two (final) outputs for the Simulink model example in Figure 3.1. In this table, we report for each executed test case and for each output, which blocks were covered (i.e., ✓) during the execution. We use the test oracle to determine which execution slices are passing and which ones are failing. The example in Table 3.1 consists of five passing and three failing execution slices. For example, the execution slice for *pOut* and $TC1$ includes *SC_Active*, *LimitP*, *IncrPres*, *PressRatioSpd*, etc, because the coverage report for *TC1* indicated that control block *SC_Active* selects (for *TC1*) the input coming from control block *LimitP*, and *LimitP* selects the input coming from *IncrPres*, and so on.

Table 3.1. Test execution slices and suspiciousness scores of model blocks using *Tarantula* for the example model of Figure 3.1.

	Test Execution Slices								**Scores**			**Overall**
	TC1		TC2		TC3		TC4					**Ranking**
Block Name	pOut	TOut	pOut	TOut	pOut	TOut	pOut	TOut	pOut	TOut	Overall	(Min-Max)
SC_Active	✓		✓		✓		✓		0.5	*NaN*	0.5	5-13
LimitP	✓		✓						0	*NaN*	0	14-20
Pmax			✓						0	*NaN*	0	14-20
IncrPres	✓								0	*NaN*	0	14-20
PressRatioSpd	✓								0	*NaN*	0	14-20
N_SC	✓								0	*NaN*	0	14-20
Pct2Val	✓								0	*NaN*	0	14-20
FlapIsClosed	✓	✓	✓	✓	✓	✓	✓	✓	0.5	0.5	0.5	5-13
FlapPosThreshold	✓	✓	✓	✓	✓	✓	✓	✓	0.5	0.5	0.5	5-13
dp		✓		✓	✓		✓		0.75	1	0.875	1- 2
p_Co		✓		✓	✓		✓		0.75	1	0.875	1- 2
pComp	✓	✓		✓	✓		✓		0.6	1	0.8	3- 4
pAdjust	✓	✓		✓	✓		✓		0.6	1	0.8	3- 4
CalcT		✓		✓		✓		✓	*NaN*	0.5	0.5	5-13
dT		✓		✓		✓		✓	*NaN*	0.5	0.5	5-13
TScaler		✓		✓		✓		✓	*NaN*	0.5	0.5	5-13
T_K2C		✓		✓		✓		✓	*NaN*	0.5	0.5	5-13
IncrP		✓		✓		✓		✓	*NaN*	0.5	0.5	5-13
T_C2K						✓		✓	*NaN*	0	0	14-20
0 C		✓		✓		✓		✓	*NaN*	0.5	0.5	5-13
Passed/Failed	Passed	Failed	Passed	Failed	Failed	Passed	Passed	Passed				

Note that our notion of test execution slice is defined over Simulink models and differs from the notion of *execution slice* defined by Agrawal [Agrawal, 1991] for programs. Specifically, an execution slice is the set of basic blocks or decisions that are executed by a test case to produce *all outputs* in programs [Agrawal, 1991]. However, test execution slices in our work are defined per test case and per output.

3.2.3 Ranking

The third step of our approach is ranking of Simulink blocks. This step takes as input test execution slices from the Slicing step, and the `PASS`/`FAIL` information for each test case and for each output from the Test Case Execution step. The output of this step is a ranked list of Simulink (atomic) blocks where each block is ranked with a suspiciousness score. The higher the suspiciousness score of a block, the higher the probability that the block has caused a failure.

To compute the suspiciousness score for a Simulink block, we use three well-known statistical formulas proposed for source code fault localization, namely, *Tarantula* [Jones et al., 2002], *Ochiai* [Abreu et al., 2007], and D^* [Wong et al., 2014]. *Tarantula* and *Ochiai* have been the subject of many experiments, and are supported by more substantial empirical evidence than other formulas [Baudry et al., 2006, Hsu et al., 2008, Jones et al., 2007, Jones and Harrold, 2005, Lucia et al., 2010, Lucia et al., 2014, Naish et al., 2011, Santelices et al., 2009, Xie et al., 2013a]. Recently, D^* has been shown to outperform 38 statistical formulas in localizing faults for programs [Wong et al., 2014]. Hence, we decided to focus on these three formulas as a representative set of the many existing statistical ranking formulas. Finally, we note that these formulas are intuitive and easy to explain. This is important as we require involvement of engineers in our experiments. Note that our technique is not tied to any particular ranking formula and can be extended to work with other statistical formulas.

Let s be a statement, and let $passed(s)$ and $failed(s)$ respectively be the number of passed and failed test cases that execute s. Let *totalpassed* and *totalfailed* represent the total number of passed and failed test cases, respectively. The suspiciousness score of s according to *Tarantula*, *Ochiai*, D^* denoted by $Score^{Ta}(s)$, $Score^{Oc}(s)$, and $Score^{D^*}(s)$, respectively, are calculated as:

$$Score^{Ta}(s) = \frac{\frac{failed(s)}{totalfailed}}{\frac{passed(s)}{totalpassed} + \frac{failed(s)}{totalfailed}}$$

$$Score^{Oc}(s) = \frac{failed(s)}{\sqrt{totalfailed \times (failed(s) + passed(s))}}$$

$$Score^{D^*}(s) = \frac{failed(s)^*}{(totalfailed - failed(s)) + passed(s)}$$

Similar to existing work [Le et al., 2014], we set * to 2 in D^*. Wong et al., [Wong et al., 2014] show that D^2 is the lowest power D^* variant that still outperforms several existing ranking formulas.

In our work, we compute the suspiciousness score of a Simulink block with respect to each individual output $o \in O$ and denote it by $Score_o$. To compute $Score_o$, we define the functions, $totalpassed_o$, $totalfailed_o$, $passed_o$, and $failed_o$ for every output $o \in O$. Based on the Test Case Execution step, we obtain the set of test execution slices and the pass/fail information for each slice. We define $totalpassed_o$, $totalfailed_o$, $passed_o(b)$ and $failed_o(b)$ for each output $o \in O$ and atomic block b as follows:

$totalpassed_o = |\{slice_{o,l} \mid \forall tc_l \in TS \land slice_{o,l} \text{ is passing}\}|$
$totalfailed_o = |\{slice_{o,l} \mid \forall tc_l \in TS \land slice_{o,l} \text{ is failing}\}|$
$passed_o(b) = |\{slice_{o,l} \mid \forall tc_l \in TS \land b \in slice_{o,l} \land slice_{o,l} \text{ is passing}\}|$
$failed_o(b) = |\{slice_{o,l} \mid \forall tc_l \in TS \land b \in slice_{o,l} \land slice_{o,l} \text{ is failing}\}|$

That is, $totalpassed_o$ and $totalfailed_o$ represent the total passing and failing test execution slices, respectively, for output o. The sets $passed_o(b)$, and $failed_o(b)$ represent the numbers of test execution slices that pass and fail, respectively, for output o, and include b. For each output $o \in O$, we define the suspiciousness score of block b for *Tarantula*, $Score_o^{Ta}(b)$, for *Ochiai*, $Score_o^{Oc}(b)$, and for D^*, $Score_o^{D^*}(b)$ as follows:

$$Score_o^{Ta}(b) = \frac{\frac{failed_o(b)}{totalfailed_o}}{\frac{passed_o(b)}{totalpassed_o} + \frac{failed_o(b)}{totalfailed_o}}$$

$$Score_o^{Oc}(b) = \frac{failed_o(b)}{\sqrt{totalfailed_o \times (failed_o(b) + passed_o(b))}}$$

$$Score_o^{D^*}(b) = \frac{failed_o(b)^*}{(totalfailed_o - failed_o(b)) + passed_o(b)}$$

Note that for a block b, $Score_o(b)$ is undefined (*NaN*) if both $passed_o(b)$ and $failed_o(b)$ are zero. This means that b has not appeared in any of the execution slices related to o.

In practice, engineers may either choose to use the scores for each output separately or combine the scores for all outputs. In particular, when there is some indication that failures in different outputs are caused by different faults, e.g., when the test execution slices of different outputs are disjoints, it is preferable to study scores separately. Otherwise, combining scores may improve the accuracy of fault localization, as in typical Simulink models a single faulty block may produce several failures in different outputs.

In our experiment in Section 3.4, we decided to combine the scores, since we want to assess the overall accuracy for all faults and outputs. We considered and experimented with several alternative ways of combining the score functions $Score_o$, and based on our experiments computing the average of the scores (see below) yielded the best experiment results. Hence, we use this method to combine the scores of the individual outputs in Section 3.4.

$$Score(b) = \frac{\sum_{o \in O \wedge Score_o(b) \neq NaN} Score_o(b)}{|\{o \in O | Score_o(b) \neq NaN\}|}$$

Having computed the scores, we now rank the blocks based on these scores. The ranking is done by putting the blocks with the same suspiciousness score in the same *rank group*. Given blocks in the same rank group, we do not know in which order the blocks are inspected by engineers to find faults. Hence, we assign a min and a max rank number to each rank group. The min rank for each rank group indicates the least number of blocks that would need to be inspected if the faulty block happens to be in this group and happens to be the first to be inspected. Similarly, the max rank indicates the greatest number blocks that would be inspected if the faulty block happens to be the last to be inspected in that group.

For example, Table 3.1 reports the *Tarantula* suspiciousness score for each block and for each of the `pOut` and `TOut` outputs as well as the mean of these two scores for the example in Figure 3.1. Note that undefined scores are shown as *NaN* cells and are not used for mean score computation. Table 3.1 also shows the block rankings obtained based on the mean scores. According to the overall ranking, the blocks `dp` and `p_Co` have the highest ranking (min rank: 1 and max rank: 2). In this example, the block `p_Co` is faulty causing both `pOut` and `TOut` to fail for different test cases. Note that if we use the scores for the `pOut` and `TOut` outputs (without averaging), four blocks `dp`, `p_Co`, `pComp`, and `pAdjust` appear in the highest rank, whereas the average ranking, which ranks two of these blocks as the highest, is more refined.

3.3 Iterative fault localization

This section describes how the approach in Figure 3.2 can be applied iteratively, allowing engineers to start with a small test oracle $\mathcal{O}$, and extend the oracle only when it is necessary. The purpose of iterative fault localization is to enable engineers to select a trade-off between the accuracy of fault localization and the cost of test oracles. The core of our iterative fault localization is a heuristic that guides engineers based on the quality of the ranking obtained at each iteration to determine whether it is worthwhile to continue fault localization with an extended test oracle or not.

Figure 3.5 shows our iterative fault localization algorithm referred to as *iSimFL*. Similar to *SimFL* (Figure 3.2), *iSimFL* takes as input a Simulink model M and a test suite TS. Since in *iSimFL*, the test oracle $\mathcal{O}$ is built incrementally, $\mathcal{O}$ is not part of its input. In addition, *iSimFL* receives two input parameters: (1) N which is the number of top most suspicious blocks that engineers typically inspect during fault localization, and (2) g which is a *coarseness threshold*. The coarseness threshold is used to determine whether a given group is too coarse or not. A rank group is too coarse if its size is larger than the maximum number of blocks that engineers can conceivably inspect (i.e., larger than g). These parameters are used in our heuristic and are domain specific. In practice, the values of these parameters are determined by archival analysis of historical fault localization data.

```
Algorithm. iSimFL

Input:  - $M$: Simulink model
        - $TS$: Test suite
        - $N$: Number of most suspicious blocks that engineers typically inspect
        - $g$ : Coarseness threshold
Output: - $L$: A ranked list of blocks

1.  Let $h$ be the hierarchy depth of $M$, let $itr = 0$, and let $\mathcal{O} = \emptyset$.
2.  do
3.    Let $\mathcal{O}'$ be the test oracle for outputs at depth $itr$, and let $\mathcal{O} = \mathcal{O} \cup \mathcal{O}'$
4.    Let $L = \{g_0, \ldots, g_m\}$ be the ranking list obtained by calling $SimFL(M, TS, \mathcal{O})$
5.    Let $L' = \{g_0, \ldots, g_k\}$ such that $|\bigcup_{0 \leq i \leq k} g_i| \geq N$ and $|\bigcup_{0 \leq i \leq k-1} g_i| < N$
6.    Let $g^*$ be the largest group in $L'$
7.    if $(|g^*| < g)$ do
8.      break;
9.    $itr{+}{+}$
10. while $(itr \leq h)$
11. return $L$
```

Figure 3.5. Iterative fault localization with *iSimFL*

As discussed in Section 2.2, Simulink models are composed of subsystems blocks that can be hierarchical. Each subsystem at each hierarchical level can have multiple outputs. We denote the *hierarchy depth* of M by h, i.e., the maximum subsystem nesting level. Model M has outputs at hierarchy depths 0 to h. For example, for the model in Figure 3.1, we have $h = 1$. The outputs `pOut` and `TOut` are at depth 0, and the outputs of `Subsystem1` and `Subsystem2` are at depth 1.

In *iSimFL*, we start at hierarchy depth zero ($itr = 0$), and iteratively build test oracle $\mathcal{O}$ such that $\mathcal{O}$ always includes the test oracle data for all the outputs from depth 0 up to depth itr. At each iteration, we call original *SimFL* with test oracle $\mathcal{O}$ (line 4) to obtain a ranked list $L = \{g_0, g_1, \ldots, g_m\}$ containing rank groups. Given a ranked list $L = \{g_0, g_1, \ldots g_m\}$, we apply our heuristic to determine whether another iteration of *iSimFL* is worthwhile or not.

Briefly, the intuition behind our heuristic is that engineers cannot effectively localize faults when the ranked list L is *coarse*, particularly within the top blocks in the list. We say a ranked list L is coarse for the top blocks, if, among the rank groups covering the top N blocks, there is a rank group whose size is larger than g (coarseness threshold). Lines 5 to 8 in Figure 3.5 implement our heuristic. If L happens to be coarse for the N top most blocks, we proceed to the next iteration where we increase itr, extend $\mathcal{O}$ to include test outputs at depth itr, and call *SimFL* with the extended test oracle. Otherwise, we terminate *iSimFL* either when L does not pass our heuristic, i.e., is not coarse (line 8), or when we reach the outputs at depth h of M (line 10).

3.4 Empirical Evaluation

This section presents our research questions and describes our industrial subjects and experimental setup, followed by the analysis of the results.

3.4.1 Research Questions

RQ1. [*SimFL*'s accuracy] *Can SimFL help localize faults by ranking the faulty blocks in the top most suspicious blocks? and what is the accuracy of SimFL for different statistical formulas*? This research question investigates whether *SimFL* can help engineers locate faulty blocks by inspecting a small subset of the model blocks. Specifically, we report the minimum and maximum number of blocks that engineers have to inspect to identify faulty blocks when they are provided with a ranked list of blocks generated by *SimFL* using *Tarantula*, *Ochiai*, and D^2. We then compare the accuracy of *SimFL* in localizing faults for these three ranking formulas.

RQ2. [Increasing test suite size] *Does increasing test suite size improve SimFL's accuracy in localizing faults?* In order to increase the spectra size, one can either increase the size of test suites or increase test oracles to include more outputs. Both require effort and have to be investigated. This research question focuses on the former to determine if increasing the size of test suites can improve the accuracy of *SimFL* in localizing faults.

RQ3. [Extending test oracle] *Does extending the set of outputs and correspondingly the test oracle to include more subsystem outputs improve the accuracy of SimFL in localizing faults?* For Simulink models, engineers often try to manually localize faults by inspecting intermediary outputs (i.e., the subsystem outputs at different hierarchy levels) in addition to final model outputs. This research question investigates the impact of increasing the number of outputs, by including subsystem outputs, on the accuracy of *SimFL* in localizing faults.

RQ4. [*iSimFL* vs *SimFL*] *How do the accuracy results of iSimFL compare with those of SimFL, and further, does iSimFL help limit the size of test oracles while improving accuracy?* Given that developing test oracles for subsystem outputs, though common and feasible, is costly, it is important to evaluate the heuristic we use in *iSimFL* to determine if it can predict when test oracle expansion is worthwhile. That is, when extending test oracles results in significant improvement in fault localization accuracy justifying the expansion overhead.

RQ5. [Impact of *iSimFL*'s parameters] *Does the performance of i*SimFL *change in a predictable way when we vary its input parameters g and N?* In **RQ4**, we compare the performance of *iSimFL* with that of *SimFL* by giving fixed values to the g and N parameters used in *iSimFL*. It is important to investigate if and how the performance of *iSimFL* is impacted when these parameters change. Specifically, for this question, we report the test oracle size required by *iSimFL* and the accuracy of *iSimFL* for different values of g and N. This data allows us (1) to determine whether the changes to the oracle size and accuracy are monotonic, and hence predictable; and (2) to identify optimal values for g and N. The optimal values of g and N are determined by comparing the results of *SimFL* and *iSimFL* and are those values that lead to a larger oracle size reduction with a negligible accuracy loss.

3.4.2 Our Industrial Subject

We use three Simulink models developed by Delphi in our experiments. These models simulate physical processes that occur inside the powertrain systems, more specifically, the combustion engine and gearbox behavior. We refer to these three models as *MS*, *MC*, and *MG*. All these three models contain different types of Simulink blocks such as switches, lookup tables, conditional blocks, integrator blocks, From/Gotos, and feedback loops. Table 3.2 shows key information about our industrial subjects. For example, Model *MS* contains 37 subsystems, 646 atomic blocks, and 596 links. The hierarchy depth is five, and the model has 12 inputs, 8 outputs at hierarchy depth zero, 8 outputs at depth one, and 7 outputs at depth two. That is, the number of outputs at depths zero and one ($\mathcal{O}_1$) is 16, and the number of outputs at depths zero, one, and two ($\mathcal{O}_2$) is 23. The outputs at depths three to five are redundant because they match those at depths one and two (e.g., in Figure 2.2, the `Subsystem2` output matches `TOut`).

Table 3.2. Key information about industrial subjects.

Model Name	# of sub-system	# of atomic blocks	# of links	# of inputs	Hierarchy Depth	# of model outputs		
						(depth 0) $\mathcal{O}_0$	(depths 0 to 1) $\mathcal{O}_1$	(depths 0 to 2) $\mathcal{O}_2$
MS	37	646	596	12	5	8	16	23
MC	64	819	798	13	7	7	11	14
MG	15	295	261	5	4	6	13	17

We asked a Delphi engineer to seed 40 realistic faults in each one of *MS* and *MC*, and 15 realistic faults in *MG*. In total, we generated 95 faulty versions (one fault per each faulty version). The faults were seeded before our experiment took place. The engineer seeded faults based on his past experience in Simulink development and, to achieve diversity in terms of the location and types of faults, we required faults of different types to be seeded in different parts of the models. We categorize the seeded faults into the following three groups: (1) *Wrong Function* which indicates a mistake in the block function type such as choosing $>$ instead of $>=$. (2) *Wrong Connection* which indicates a wrong link between two blocks. For example, engineers may connect the signal A to input 2 instead of input 1 of a block. Note that if the data type of signal A and input 2 does not match, Simulink reports a syntax error. Hence, this fault refers to cases where the types match, but the connection is still wrong. (3) *Wrong Value*, indicating a wrong value in a constant Simulink block or a wrong threshold value in a Simulink control block.

The above classification of faults does not include Stateflow [MathWorks, b], which is the state machine notation of Simulink. This is because (1) our industrial subjects do not include any Stateflows, and (2) we would need to adapt slicing to Stateflow, which is out of the scope of this dissertation. Further, Simulink models may fail due to the wrong configuration of the simulator, e.g., a wrong step size. In our work, we focus on handling failures caused by faults applied to the model and not those that are due to the wrong configuration of the simulator.

Finally, we note that our industrial subjects are representative in terms of size and complexity among Simulink models developed at Delphi. Our industrial subject models include about ten times more blocks than the publicly available Simulink models from the Mathworks model repository [MathWorks, a]. In addition, most publicly available Simulink models are small exemplars created for the purpose of training for which realistic faults are not available. Hence, we chose to perform our experiments exclusively on industrial subjects for which *realistic* faults could be obtained from an experienced engineer.

3.4.3 Experiment Settings

In addition to a Simulink model, which is discussed in Section 3.4.2, *SimFL* requires as input a test suite and a test oracle which are discussed below, along with the experiment design and evaluation metrics.

Test Suite. In this chapter, we generated test suites using Adaptive Random Testing [Chen et al., 2005]. In our experiment, we were provided with the valid ranges of input signals of our industrial subjects. Adaptive random testing is a black box and lightweight test generation strategy that distributes test cases evenly within the input space (i.e., the valid ranges), and therefore, helps ensure diversity among test cases.

Test Oracle. In practice, the development of test oracles is largely manual and out of scope of this dissertation. In our experiment, we chose to use a fault-free version of our industrial subject model for the oracle information to automate our large-scale and time-consuming experiments. Note that the Simulink models used in our experiment, when provided with constant input signals, are expected to stabilize and eventually converge to a constant output signal (see Figure 2.3(b)). If the output signal does not stabilize within a sufficiently large simulation time interval, we mark that as a failure. In this case study, we followed the suggestion from Delphi engineers and set the simulation time (and thus, the moment at which we measure the output signal) to 10 seconds. For each output, to determine if a test case passes or fails, we compared the values of that output from the faulty Simulink model with the fault-free Simulink model at the end of a 10-sec simulation. If they matched, we marked the output as `PASS`, and otherwise, as `FAIL`.

Experiments. We perform five separate experiments, referred to as *EXP1*, *EXP2*, *EXP3*, *EXP4*, and *EXP5* to answer our research questions. In our experiments, we consider three different sets of outputs and their corresponding test oracles: (1) test oracle $\mathcal{O}_0$ for the model outputs at depth zero, (2) test oracle $\mathcal{O}_1$ for the outputs at depth zero and one, and (3) test oracle $\mathcal{O}_2$ for the outputs at depths zero, one, and two. Table 3.2 shows the sizes of test oracles $\mathcal{O}_0$ to $\mathcal{O}_2$ for each industrial subject.

To answer **RQ1**, we perform experiment *EXP1* where we apply *SimFL* (Figure 3.2) using *Tarantula*, *Ochiai*, and D^2 to our 95 faulty models with a test suite size of 200 and with the smallest test oracle ($\mathcal{O}_0$). Note that the size selected for test suites was based on typical practice at Delphi given test budget constraints and the cost of oracles. For **RQ2**, we perform experiment *EXP2* where we apply *SimFL* using *Tarantula*, *Ochiai*, and D^2 to our 95 faulty models with the test oracle $\mathcal{O}_0$ and with nine different test suites of varying size: 200, 300, 400, 500, 600, 700, 800, 900, and 1000. We start from the test suite with 200 test cases and augment the test suites by adding (100) more test cases generated using Adaptive Random Testing. For **RQ3**, we perform experiment *EXP3* where we apply *SimFL* using *Tarantula*, *Ochiai*, and D^2 to our 95 faulty models with a test suite size of 200 and with test oracles $\mathcal{O}_1$ and $\mathcal{O}_2$. Based on the results of *EXP1*, *EXP2*, and *EXP3*, we select one statistical formula to be used by *iSimFL* in *EXP4* to *EXP5*. For **RQ4**, we perform experiment *EXP4* where we apply *iSimFL* (Figure 3.5) to our 95 faulty models with a test suite size of 200 and rely on the heuristic used in *iSimFL* to determine how many iterations are required for each faulty model. For the parameters N and g used in *iSimFL*, we set their values based on our experience and discussions with domain experts. Specifically, we set $N = 15$ because engineers, when provided with a ranked list of blocks, are able to typically and routinely inspect the top 15 blocks. Further, for each faulty model, we set g to 6% of the size of the model. Finally, for **RQ5**, we perform experiment *EXP5* where we apply *iSimFL* with different values of N and g to our 95 faulty models with a test suite size of 200. *EXP5* consists of two parts i.e., *EXP5a* and *EXP5b*. For *EXP5a*, we apply *iSimFL* where we fix N to 15 and vary the value of g to 1%, 2%, ..., 10% of the model size. For *EXP5b*, we apply *iSimFL* where we fix g to 6% of the model size and vary the value of N to 5, 10, ..., 25.

Evaluation Metrics. Assuming that engineers inspect block rankings generated by *SimFL* or *iSimFL* to find faults, we evaluate the accuracy of *SimFL* and *iSimFL* using the following metrics from the fault localization literature [Cleve and Zeller, 2005, Jones and Harrold, 2005, Liu et al., 2005, Lucia et al., 2014, Parnin and Orso, 2011, Renieris and Reiss, 2003]: The *percentage of blocks inspected* to find faults, the *absolute number of blocks inspected* to find faults, and the *proportion of faults localized* when engineers inspect fixed numbers of the top most suspicious blocks.

For the *absolute number of blocks inspected* to find faults, we consider the min and the max ranks of the rank group that contains the faulty block. For the *percentage of blocks inspected* to find faults, we divide the absolute number of blocks inspected (both for the min and the max ranks) by the total number of blocks. The *proportion of faults localized* is the proportion of localized faults over the total number of faults when engineers inspect a fixed number of the top most suspicious blocks from a ranked list.

3.4.4 Experiment Results

In this section, we address our research questions based on our experiment results.

RQ1. [*SimFL*'s accuracy] To answer this question, we performed *EXP1* described in Section 3.4.3. We evaluate *SimFL*'s accuracy in localizing faults in terms of the percentage and the absolute number of blocks inspected, and the proportion of faults localized, as follows:

Percentage and absolute number of blocks inspected. In Table 3.3, we show the percentages and absolute numbers of blocks that engineers need to inspect when they use *SimFL* with the smallest test oracle (i.e. $\mathcal{O}_0$) for three formulas i.e., *Tarantula*, *Ochiai*, and D^2. For all 95 models, when using *SimFL* with $\mathcal{O}_0$ and *Tarantula* as the statistical formula, engineers need to inspect, on average, at least 14 and at most 63 blocks (i.e., 2.1% - 8.9%). Similarly, when using *SimFL* with *Ochiai* as the statistical formula, engineers need to inspect, on average, at least 23 and at most 62 blocks (i.e., 3.1% - 8.8%), and when using *SimFL* with D^2, engineers need to inspect, on average, at least 16 and at most 56 (i.e., 2.3% - 7.9%).

Table 3.3. Average percentage and absolute number of blocks inspected using *SimFL* with $\mathcal{O}_0$ for *Tarantula*, *Ochiai*, and D^2.

Model name	min.#(%) - max.#(%) for *SimFL* with $\mathcal{O}_0$		
	Tarantula	*Ochiai*	D^2
MS	13 (2.1%) - 46 (7.1%)	14(2.2%) - 43(6.7%)	11(1.6%) - 40(6.1%)
MC	19 (2.4%) - 96 (11.7%)	39(4.7%) - 98(12%)	26(3.2%) - 86(10.5%)
MG	4 (1.4%) - 18(6%)	5(1.6%) - 18(6%)	4(1.4%) - 18(6.0%)
All models	14 (2.1%) - 63 (8.9%)	23 (3.1%) - 62 (8.8%)	16 (2.3%) - 56 (7.9%)

Proportion of faults localized. In Figures 3.6, 3.7, and 3.8, we present the proportion of faults localized when engineers inspect a fixed number of the most suspicious blocks in the rank lists generated by *SimFL* with three statistical formulas (i.e., *Tarantula*, *Ochiai*, and D^2) for *MS*, *MC*, and *MG*, respectively. In each figure, the solid line shows the maximum proportion of faults localized, and the dashed line shows the minimum proportion of faults localized.

For 40 faulty versions of *MS* (see Figure 3.6), by inspecting the top 10% of most suspicious blocks (i.e., 65 blocks), engineers can locate at most 95% and at least 78% of the faults when *SimFL* with *Tarantula* is used; at most 93% and at least 78% of the faults when *SimFL* with *Ochiai* is used; and at most 95% and at least 83% of the faults when *SimFL* with D^2 is used. For 40 faulty versions of *MC* (see Figure 3.7), by inspecting the top 10% of most suspicious blocks (i.e., 82 blocks), engineers can locate at most 85% and at least 33% of the faults when *SimFL* with *Tarantula* is used; at most 80% and at least 33% of the faults when *SimFL* with *Ochiai* is used; and at most 85% and at least 35% of the faults when *SimFL* with D^2 is used. For 15 faulty versions of *MG* (see Figure 3.8), by inspecting the top 10% of most suspicious blocks (i.e., 30 blocks), engineers can locate at most 100% and at least 93% of the faults when *SimFL* with *Tarantula* and D^2 are used; and at most 100% and at least 87% of the faults when *SimFL* with *Ochiai* is used.

Note that, for all of the three formulas (i.e., *Tarantula*, *Ochiai*, and D^2), the results of using *SimFL* for *MC* is not as good as the results for the other two models. This is because, compared to *MS* and *MG*, *MC* includes a larger number of lookup tables, integrator blocks, unit convertors, and trigonometry and logarithmic functions that may potentially reduce or mask data discrepancies, and hence, impact the number of observed failures for outputs at depth zero. As a result, the fault localization results for *MC* when we focus on the outputs in $\mathcal{O}_0$ are less accurate compared to the results for *MS* and *MG*.

Comparing with the state-of-the-art. Since no studies on fault localization for Simulink models are reported, we briefly report the results obtained from applying statistical debugging approaches (with various ranking formulas) to source code implemented in C or Java. We note that, like our work, the approaches discussed here assume that the code under analysis has a single fault.

Comparing the percentage of blocks inspected to localize faults in programs, on average, developers need to inspect at most around 20% of their code (i.e., program blocks) to localize faults [Cleve and Zeller, 2005, Liu et al., 2005, Lucia et al., 2014, Renieris and Reiss, 2003], while *SimFL*, on average, requires at most around 8% (i.e., 8.9% for *SimFL* with *Tarantula*, 8.8% for *SimFL* with *Ochiai*, and 7.9% for *SimFL* with D^2) of the model blocks to be inspected to find faults. Comparing the proportion of faults localized, assuming that developers only inspect the top 10% of the most suspicious code elements, on average, the minimum percentage of faults localized is less than 55% [Cleve and Zeller, 2005, Liu et al., 2005, Lucia et al., 2014, Renieris and Reiss, 2003]. When engineers inspect 10% of the top most suspicious blocks returned by *SimFL*, on average, the minimum percentage of faults localized is around 60% (i.e., 58/95 for *SimFL* with *Tarantula* 57/95 for *SimFL* with *Ochiai*, and 61/95 for *SimFL* with D^2).

While source code debugging and Simulink model debugging have major differences, the above comparison shows that our results are promising and statistical debugging for Simulink models is potentially useful. We note that while inspecting 10% of software code may indeed require developers to review tens or hundreds of KLOC, 10% of a typical Simulink model is often less than 100 blocks. Moreover, engineers are often able to conceptually trace Simulink blocks to abstract functions and concepts, making it easier for them to determine whether an individual block is faulty or not.

Comparing Tarantula*,* Ochiai*, and* D^2. The above results show that the accuracy of ranking results obtained by these three formulas are considerably close. Based on Table 3.3, the percentage and absolute numbers of blocks inspected using *SimFL* with $\mathcal{O}_0$ for *Tarantula*, *Ochiai*, and D^2 are considerably close. The average maximum percentages of blocks inspected corresponding to the three formulas range from 7.9% to 8.9%. Figure 3.9 shows the comparison of the proportions of faults localized for all 95 faulty versions when using *SimFL* with *Tarantula*, *Ochiai*, and D^2. Based on Figure 3.9, the minimum proportions of faults localized when using the three formulas are close, in particular when engineers inspect the top 20 blocks. Further, when engineers inspect more than 20 blocks, the variations in the minimum proportions of faults localized across the three formulas are less than 7%, and hence, not substantial.

In summary, the answer to **RQ1** is that, on average, *SimFL* is able to rank the faulty blocks as the most suspicious blocks that should be inspected by engineers. Further, the accuracies of *SimFL* using *Tarantula*, *Ochiai*, and D^2 in localizing faults in Simulink models are not substantially different.

RQ2.[Increasing test suite size] To answer this question, we performed *EXP2*. We observed that the maximum number of blocks that engineers need to inspect to find the fault in each faulty model remains almost constant as we apply *SimFL* with *Tarantula*, *SimFL* with *Ochiai*, and *SimFL* with D^2

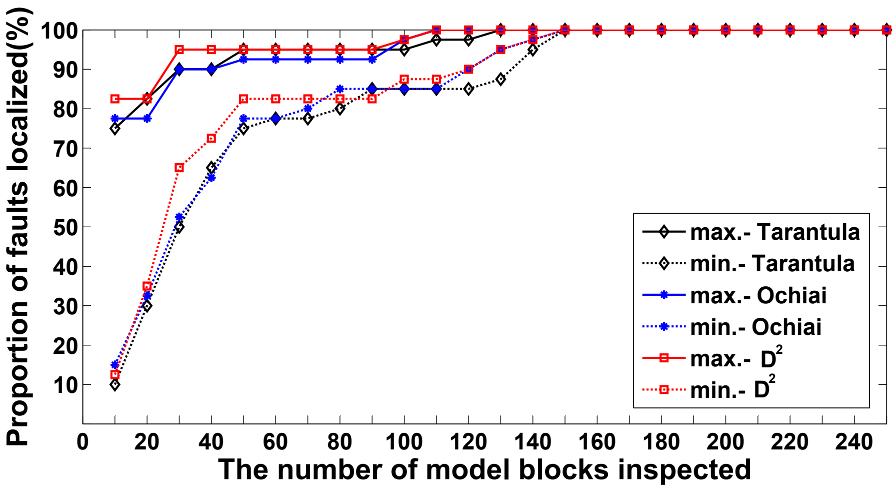


Figure 3.6. Proportion of faults localized for *MS* using *SimFL* with $\mathcal{O}_0$.

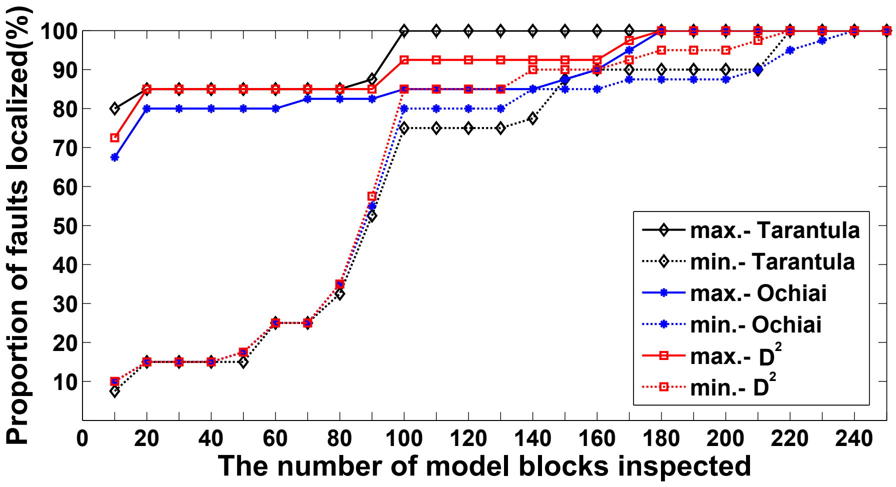


Figure 3.7. Proportion of faults localized for *MC* using *SimFL* with $\mathcal{O}_0$.

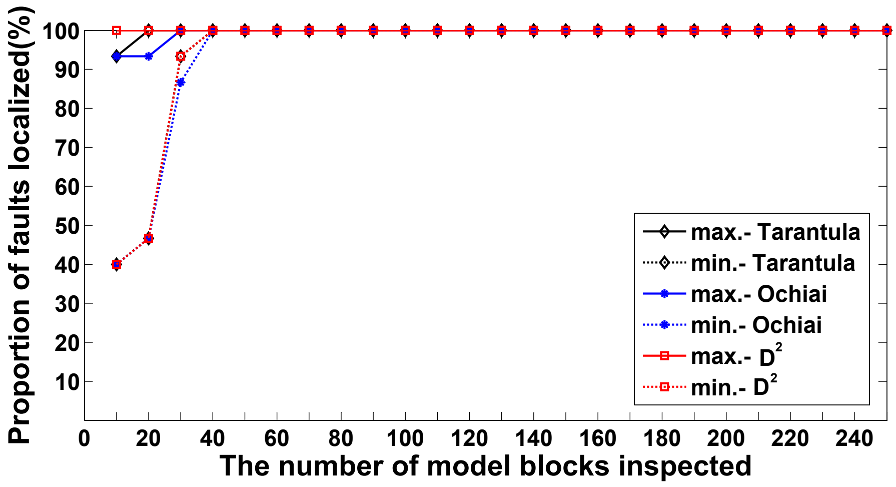


Figure 3.8. Proportion of faults localized for *MG* using *SimFL* with $\mathcal{O}_0$.

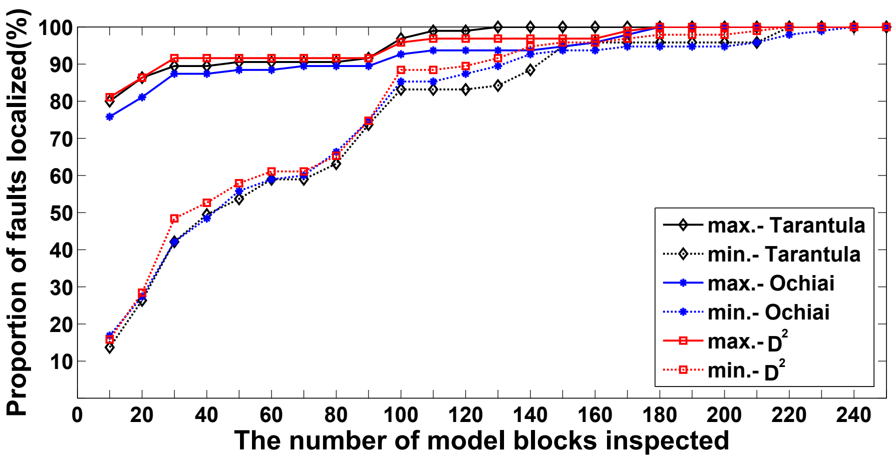


Figure 3.9. Proportion of faults localized for *all* models using *SimFL* with $\mathcal{O}_0$.

using test suite sizes of 200, 300, ..., 1000. For all the 95 faulty models and for these three formulas, changes in the rankings of the faulty blocks are negligible as we apply *SimFL* with different test suite sizes. Specifically, for the three formulas, the differences on the maximum percentages of blocks inspected are less than 1.8% as we increase the test suite size. Note that we start with a test suite with size 200 because this size is realistic and comparable to test suite sizes used for Simulink models in Delphi.

To explain why the rankings of the faulty blocks remain almost constant, we introduce the notion of *Coincidentally Correct Test cases (CCT)* [Wang et al., 2009]. CCTs are test execution slices that execute faulty blocks but do not result in failure. CCTs are likely to occur in Simulink models because these models often contain various mathematical function blocks that may reduce or mask data discrepancies, resulting in passing test execution slices that exercise faulty blocks. We note that based on the *Tarantula*, *Ochiai*, and D^2 formulas, and given that our faulty models include a single faulty block, the following hold: (1) The *Tarantula* scores of faulty blocks depend on the proportion of CCTs over the total number of passing test execution slices [Wang et al., 2009], (2) the *Ochiai* scores of faulty blocks depend on the proportion of total failing test execution slices over the number of test execution slices that cover the faulty blocks, and (3) the D^2 scores of faulty blocks depend on the proportion of total failing test execution slices over the number of CCTs.

In our experiments, we observed that as we increase the test suite size: (1) the proportion of CCT over all passing test execution slices remains almost constant (i.e., changes in this proportion are less than 1%), (2) the proportion of total failing test execution slices over the total number of test execution slices that cover the faulty blocks also remains almost constant with an average difference of 0.004, and (3) the changes in the proportion of total failing test execution slices over the number of CCTs are negligible (i.e., the average difference is 0.22). Hence, increasing the test suite size has no notable impact on the rank of faulty blocks for none of these three formulas.

In summary, the answer to **RQ2** is that increasing the size of test suites, above what can be considered a typical size in our application context, does not make any significant changes in *SimFL*'s accuracy.

RQ3.[Extending test oracles] To answer this question, we performed *EXP3*. In Figures 3.10(a) to 3.10(f), we show the maximum percentages of blocks required to be inspected for 58 out of 95 faulty models when *SimFL* with *Tarantula* is applied with test oracles $\mathcal{O}_0$, $\mathcal{O}_1$, and $\mathcal{O}_2$. The results for the other 37 models are not shown, as *SimFL* with $\mathcal{O}_0$ already performs well, i.e., on average, the maximum percentage of blocks inspected is 4.4% and by extending test oracles of those 37 models, the accuracy only slightly improves or remains the same.

Among the 58 models under consideration, *SimFL* with $\mathcal{O}_0$ performs reasonably well for 20 models (Figures 3.10(a) and 3.10(b)) as the maximum percentage of blocks inspected for these models is less than 10%. We still chose to show the results for these 20 models because these results are used to answer **RQ4** as well. For the other 38 models (i.e., models in Figures 3.10(c) to 3.10(f)), *SimFL* with $\mathcal{O}_0$ requires engineers to inspect more than 10% of the blocks in order to locate faults (between 10.2% and 26.6%).

For faulty models shown in Figure 3.10(a), extending test oracles from $\mathcal{O}_0$ to $\mathcal{O}_2$ improves *SimFL*'s accuracy slightly (i.e., up to 3%) for 13 models, while for the other three models (i.e., *MS39*, *MG3*, and *MG9*), *SimFL*'s accuracy remains the same. For 29 faulty models (Figures 3.10(b) to 3.10(d)), extending test oracles from $\mathcal{O}_0$ to $\mathcal{O}_1$ notably improves *SimFL*'s accuracy. Specifically, on average, the maximum percentage of blocks inspected reduces to 1.8%, 3%, and 10% for the models in Figures 3.10(b), 3.10(c), and 3.10(d), respectively. However, for these models, the accuracy improves slightly or remains the same when we extend the oracle to $\mathcal{O}_2$. On the contrary, for the 10 models as shown in Figure 3.10(e), extending the oracle to $\mathcal{O}_1$ improves *SimFL*'s accuracy slightly, but extending the oracle to $\mathcal{O}_2$ notably improves the accuracy to, on average, 4%. Note that extending the test oracle could potentially increase the number of failing execution slices that are useful for localizing faults. In Table 3.4, we show the minimum and maximum numbers of failing execution slices for all the faulty versions of *MS*, *MC*, and *MG*, as we extend the test oracle from $\mathcal{O}_0$ to $\mathcal{O}_2$. For the large difference between the minimum and maximum, we can see that certain faults are much easier to detect than others and hence they result in many more failing execution slices. Based on Table 3.4, the minimum and maximum numbers of failing execution slices increase or remain the same as we extend the test oracle from $\mathcal{O}_0$ to $\mathcal{O}_2$.

In contrast to the above models, where *SimFL*'s accuracy either improves or stays the same as we expand the oracle, for *MS32*, *MS33*, and *MS14* (Figure 3.10(f)), *SimFL* may fare worse as we extend the oracle. For *MS32* and *MS33*, the maximum percentages of blocks that engineers need to inspect decrease to below 10% (i.e., 5.8%) when going from $\mathcal{O}_0$ to $\mathcal{O}_1$, but these percentages increase to above 10% (i.e., 15.7% and 10.9%) again when $\mathcal{O}_2$ is used. As for *MS14*, *SimFL* fares worse when we extend the oracle from $\mathcal{O}_0$ to $\mathcal{O}_1$. But after extending the oracle to $\mathcal{O}_2$, we observe a high improvement (i.e., 8.3%).

To explain why test oracle expansion does not always improve accuracy, we note that as we extend the size of test oracles, either the number of CCTs increases or stays the same. In the latter case, *SimFL*'s accuracy either improves or remains the same because none of the new *passing* test execution slices exercise faults, and hence, the block rankings either stay the same or become more accurate. In the former case, however, *SimFL*'s accuracy is unpredictable and may even decrease. Our experiment data confirms this intuition. For the cases where *SimFL*'s accuracy declines as we increase the spectra size, i.e., for *MS32* and *MS33* (from $\mathcal{O}_1$ to $\mathcal{O}_2$) and for *MS14* (from $\mathcal{O}_0$ to $\mathcal{O}_1$), the size of CCT increases.

Nevertheless, we note that for all but three faulty models, *SimFL*'s accuracy improves or remains

Table 3.4. Number of failing execution slices based on test oracles $\mathcal{O}_0$, $\mathcal{O}_1$, and $\mathcal{O}_2$.

Model Name	Test suite size	# of failing slices (min. $\sim$ max.)		
		Test oracle $\mathcal{O}_0$	Test oracle $\mathcal{O}_1$	Test oracle $\mathcal{O}_2$
MS	200	$6 \sim 1009$	$34 \sim 1796$	$47 \sim 2461$
MC	200	$8 \sim 1010$	$8 \sim 1390$	$8 \sim 1390$
MG	200	$34 \sim 249$	$92 \sim 467$	$102 \sim 467$

the same as we extend the test oracles to include more subsystem outputs. When using *SimFL* with $\mathcal{O}_2$, on average, engineers need to inspect, for *MS*, at least 1.7% and at most 4.0% of model blocks (i.e., 11 to 26 blocks); for *MC*, at least 1.1% and at most 4.1% of model blocks (i.e., 9 to 34 blocks); and for *MG*, at least 1.4% and at most 3.4% of model blocks (i.e., 4 to 10 blocks). On average, for all 95 faulty models and using *SimFL* with $\mathcal{O}_2$, engineers need to inspect at least 1.4% and at most 4% of model blocks (i.e., 9 to 27 blocks) , which is less than the results for *SimFL* with $\mathcal{O}_0$ (i.e., on average, at least 2.1% and at most 8.9% of model blocks). Furthermore, using *SimFL* with $\mathcal{O}_2$, by inspecting only the top 10% of most suspicious blocks, engineers are able to find at least 91 out of 95 faults, which is 33 more faults compared to using *SimFL* with $\mathcal{O}_0$.

When we extend the test oracle to $\mathcal{O}_2$, the accuracy of *SimFL* with *Ochiai* and *SimFL* with D^2 is similar to the accuracy of *SimFL with Tarantula.* Specifically, for all the 95 faulty models and for *SimFL* with *Ochiai*, the number of blocks inspected decreases to, on average, at least 1.8% and at most 4% of the model blocks (i.e., 13 to 27 blocks). Similarly, for *SimFL* with D^2, the number of blocks inspected decreases to, on average, at least 1.5% and at most 3.6% of the model blocks (i.e., 10 to 24 blocks).

In summary, the answer to **RQ3** is that extending test oracles by including more outputs at lower hierarchy levels may or may not improve *SimFL*'s accuracy in localizing faults on a specific model. But overall, oracle extension leads to the detection of significantly more faults.

Since, as shown in **RQ1** to **RQ3**, there is no significant differences in the accuracies of*Tarantula*, *Ochiai*, and D^2 in localizing faults in Simulink models, we answer **RQ4** and **RQ5** based on *iSimFL* results with one of these formulas. In particular, we report the results for *iSimFL* with *Tarantula* as its results are representative for the other two formulas.

RQ4.[*iSimFL* vs *SimFL*] To answer this question, we performed *EXP4*. Our experiment shows that for 37 out of 95 models (not shown in Figure 3.10), *iSimFL* only performed one iteration before it terminates. That is, the loop in Figure 3.5 was executed only once and with oracle $\mathcal{O}_0$ for these 37 models. The maximum percentages of blocks inspected for these 37 models with $\mathcal{O}_0$ are reasonably low (4.4% on average) and hence, as *iSimFL* correctly predicted, oracle expansion is not necessary. The results of *EXP4* for the other 58 models are shown in Figures 3.10(g) to 3.10(l).

For 16 faulty models as shown in Figure 3.10(g), *iSimFL* extends test oracles although the maximum percentages of blocks inspected using $\mathcal{O}_0$ are already good (4.2% on average) and oracle expansion does not lead to a substantial improvement. For these models, the *iSimFL* heuristic still extended $\mathcal{O}_0$ because there were some coarse groups (with size larger than g) below the faulty block but within the top N blocks. Specifically, for eight of these 16 models, *iSimFL* extends test oracle to $\mathcal{O}_1$, and for the other eight models (i.e., *MS3*, *MS12*, *MS34*, *MS36*, *MC2*, *MC29*, *MC31*, *MC33*), *iSimFL* extends

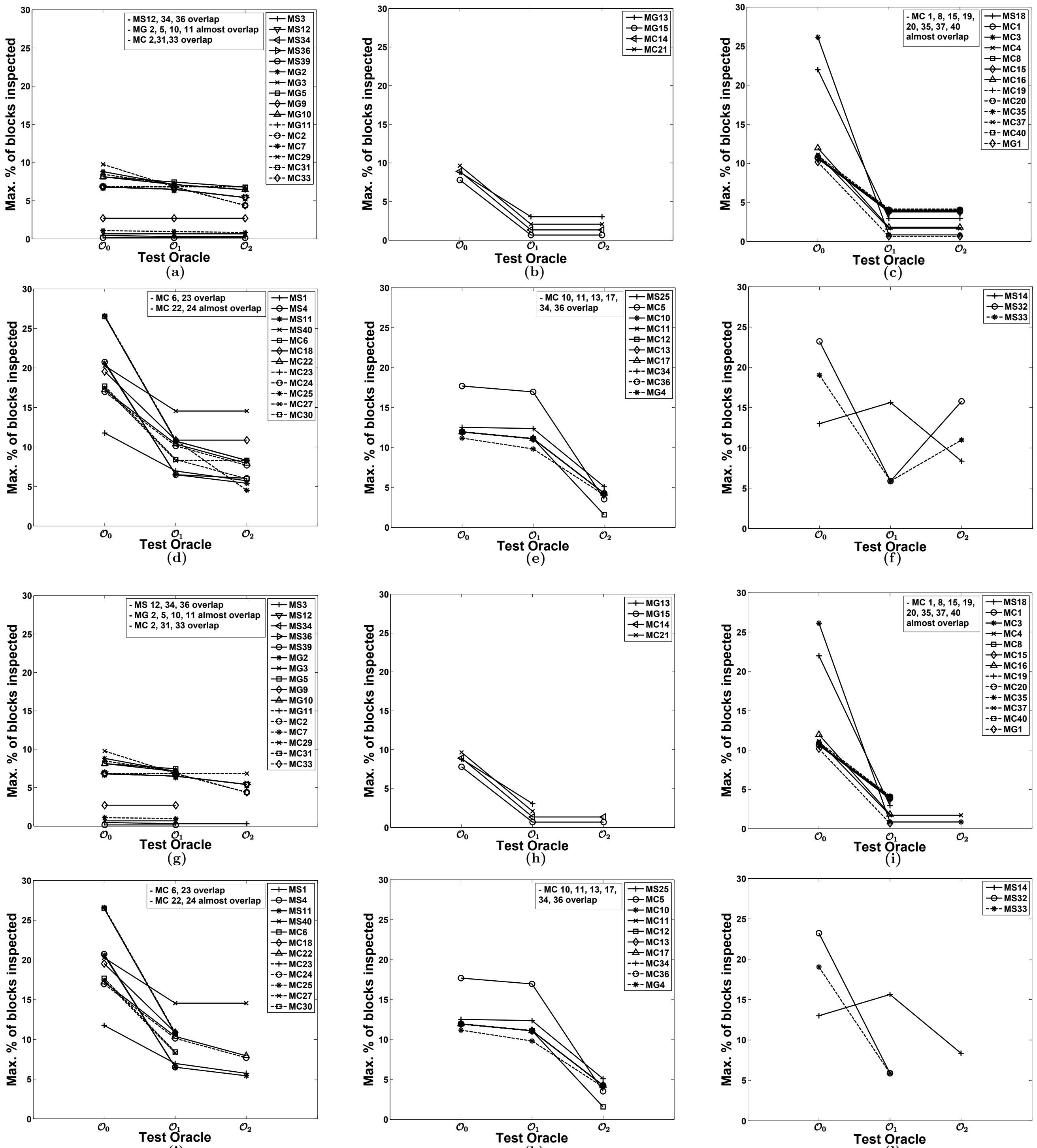


Figure 3.10. Maximum percentage of blocks that need to be inspected to find faults for *SimFL* with *Tarantula* and test oracles $\mathcal{O}_0$, $\mathcal{O}_1$, and $\mathcal{O}_2$ and for *iSimFL* with *Tarantula*: (a) *SimFL*'s accuracy improves slightly or remains the same as we extend the oracle, (b-e) *SimFL*'s accuracy improves notably as we extend the oracle, (f) *SimFL*'s accuracy is unpredictable as we extend the oracle, and (g-l) *iSimFL*'s accuracy for those models where, according to the *iSimFL*'s heuristic, oracle expansion is required.

test oracle to $\mathcal{O}_2$.

For four faulty models as shown in Figure 3.10(h), the maximum percentages of blocks inspected using $\mathcal{O}_0$ are within an acceptable range (8.8% on average). Nevertheless, extending test oracles to $\mathcal{O}_1$ is still beneficial. For these models, *iSimFL* correctly extends test oracles to $\mathcal{O}_1$. By doing so, on average, the maximum percentage of blocks inspected notably decreases from 8.8% to 1.8% of the model blocks. However, for *MC14* and *MG15*, *iSimFL* unnecessarily extends the oracles to $\mathcal{O}_2$ while the maximum percentage of blocks inspected remains the same.

For the other 38 models (Figures 3.10(i) to 3.10(l)), the maximum percentages of blocks inspected using $\mathcal{O}_0$ are considerably high (15.5% on average). For 34 of the 38 models, *iSimFL* correctly extends oracles which substantially decreases the maximum percentage of blocks inspected. Specifically, *iSimFL* correctly performs two iterations (with $\mathcal{O}_0$ and $\mathcal{O}_1$) for 20 models and three iterations (with $\mathcal{O}_0$, $\mathcal{O}_1$, and $\mathcal{O}_2$) for 14 models. For these 34 models, *iSimFL* continues extending the oracle either until its accuracy improves and fall below 10%, or until no further extension is possible. Note that only one model (i.e., *MS40*) falls in the latter group. Further, for four models (i.e., *MS1*, *MS11*, *MC3*, and *MC4* (Figure 3.10(i))), *iSimFL* correctly predicts that extending oracles to $\mathcal{O}_1$ is beneficial, though *iSimFL* additionally and unnecessarily extends the oracles to $\mathcal{O}_2$ while the accuracy remains the same or does not substantially improve.

In summary, oracle extension is not necessary for 53 out of 95 models. For the other 42 models where it is necessary (i.e., leads to considerable improvement in accuracy), 28 models need to extend the test oracle up to depth one (i.e., $\mathcal{O}_1$), and 14 models require to extend the test oracle up to depth two (i.e., both $\mathcal{O}_1$ and $\mathcal{O}_2$).

The *iSimFL* heuristic was able to correctly identify 37 out of 53 models that do not need oracle extension and correctly identify all models (i.e., 42) that require oracle extension. Among these 42 models, the *iSimFL* correctly predict the oracle extension depth for 36 models. For the other six models (i.e., *MS1*, *MS11*, *MC3*, *MC4*, *MC14*, and *MG15*), *iSimFL* correctly extends test oracles to $\mathcal{O}_1$, but *iSimFL* unnecessarily extends test oracles further to $\mathcal{O}_2$. Further, using *iSimFL*, the average oracle size for each model is about 12 and therefore lower compared to the size of $\mathcal{O}_2$ (23 for *MS*, 14 for *MC*, and 17 for *MG*). Finally, *iSimFL* was able to properly handle the three cases discussed in **RQ3** where oracle extension caused the accuracy to decline (Figure 3.10(f)). Specifically, for *MS32* and *MS33*, *iSimFL* stops after applying $\mathcal{O}_1$, whereas for *MS14*, it goes all the way to $\mathcal{O}_2$.

Table 3.5 shows the minimum and maximum numbers (and percentages) of blocks inspected for each industrial subject, comparing *SimFL* with $\mathcal{O}_0$, *SimFL* with $\mathcal{O}_2$ (i.e., extending all oracles), and *iSimFL*. Specifically, after applying *iSimFL* to our 95 faulty models, we obtained the following values for our evaluation metrics:

Percentage and absolute number of blocks inspected. For all models, using *iSimFL*, engineers need to inspect, on average, at least 1.3% and at most 4.4% of model blocks. As shown in Table 3.5, these results are comparable to those obtained by *SimFL* with $\mathcal{O}_2$ and are better than those obtained by *SimFL* with $\mathcal{O}_0$.

Proportion of faults localized. Using *iSimFL*, engineers can find at least 90 out of 95 faults (i.e., 95%) when only the top 10% of most suspicious blocks are inspected. *iSimFL* is able to locate a

Table 3.5. Average of minimum and maximum numbers of blocks inspected and test oracle sizes when using *SimFL-Tarantula* with $\mathcal{O}_0$, *SimFL-Tarantula* with $\mathcal{O}_2$, and *iSimFL-Tarantula*.

Model	*SimFL* with $\mathcal{O}_0$	*SimFL* with $\mathcal{O}_2$	*iSimFL*
name	min. #(%) - max. #(%) ($\lvert\mathcal{O}_0\rvert$)	min. #(%) - max. #(%) ($\lvert\mathcal{O}_2\rvert$)	min. #(%) - max. #(%) (Avg.$\lvert\mathcal{O}\rvert$)
MS	13 (2.1%) - 46 (7.1%) (8 outputs)	11 (1.7%) - 26 (4.0%) (23 outputs)	9 (1.5%) - 29 (4.5%) (12 outputs)
MC	19 (2.4%) - 96 (11.7%) (7 outputs)	9 (1.1%) - 34 (4.1%) (14 outputs)	10 (1.2%) - 37 (4.5%) (12 outputs)
MG	4(1.4%) - 18(6.0%) (6 outputs)	4 (1.4%) - 10 (3.4%) (17 outputs)	4 (1.4%) - 11 (3.7%) (11 outputs)

similar number of faults compared to *SimFL* with $\mathcal{O}_2$ (i.e., 90 vs. 91).

In summary, the answer to **RQ4** is that the accuracy of *iSimFL* is similar to the accuracy of *SimFL* with $\mathcal{O}_2$, while the average test oracle size for *iSimFL* is 12 compared to a larger size for O2 (12 vs. 23 for MS, 12 vs. 14 for MC, and 11 vs. 17 for MG). That is, *iSimFL* achieves the same accuracy as *SimFL* with $\mathcal{O}_2$ using smaller test oracles. Further, *iSimFL*, with an average oracle size of 12, yields a significant improvement in accuracy over *SimFL* with $\mathcal{O}_0$, which has an average oracle size of 7. That is, *iSimFL* extends only by five outputs the oracle $\mathcal{O}_0$.

RQ5.[Impact of *iSimFL*'s parameters] To answer this question, we performed *EXP5a* and *EXP5b* as described in Section 3.4.3. We evaluated the impact of changes in the values of N and g parameters of *iSimFL* on the average accuracy and the average oracle size extension of *iSimFL*. The reference for comparison is *SimFL* with the maximum oracle ($\mathcal{O}_2$). Specifically, we want to know, when changing N and g, how the average accuracy and the average oracle size of *iSimFL* fare compared to the accuracy and the test oracle size of *SimFL* with $\mathcal{O}_2$.

Figures 3.11 and 3.12 show the results of these experiments: In Figure 3.11, we show the results of *EXP5a* where N is fixed at 15 and we vary the value of g from 1% to 10% of model blocks. Specifically, Figure 3.11(a) shows the average reduction in the oracle size required by *iSimFL* compared to the size of $\mathcal{O}_2$ for *MS*, *MC*, and *MG*, and Figure 3.11(b) shows the average loss in the accuracy of *iSimFL*, which tries to use smaller oracles than $\mathcal{O}_2$, compared to the accuracy of *SimFL* with $\mathcal{O}_2$ for *MS*, *MC*, and *MG*. For example, based on the results in these figures, by applying *iSimFL* to *MS* and when g is set to 3% of the size of *MS*, the average accuracy of the rankings generated by *iSimFL* is around 2 (blocks) less than the average accuracy of rankings obtained by *SimFL* with $\mathcal{O}_2$ (see Figure 3.11(b)). But *iSimFL* obtains these rankings with an oracle that contains on average seven less outputs compared to $\mathcal{O}_2$ (see Figure 3.11(a)). In Figure 3.12, we show the results of *EXP5b* where g is set to 6% of the size of the underlying models and N is set to 5, 10, 15, 20, and 25. Similar to Figure 3.11, Figure 3.12(a) shows the average reduction in the oracle size required by *iSimFL* compared to the size of $\mathcal{O}_2$ for *MS*, *MC*, and *MG*, and Figure 3.12(b) shows the average loss in the accuracy of *iSimFL* compared to the accuracy of *SimFL* with $\mathcal{O}_2$ for *MS*, *MC*, and *MG*.

As shown in Figure 3.11, for $N = 15$, as the value of g increases, *iSimFL* extends test oracles less (i.e., the difference between the oracle size required by *iSimFL* and size of $\mathcal{O}_2$ increases), while the accuracy of ranking results mostly decreases (i.e., engineers on average have to inspect more blocks to find the fault compared to the number of blocks that they need to inspect when *SimFL* with $\mathcal{O}_2$

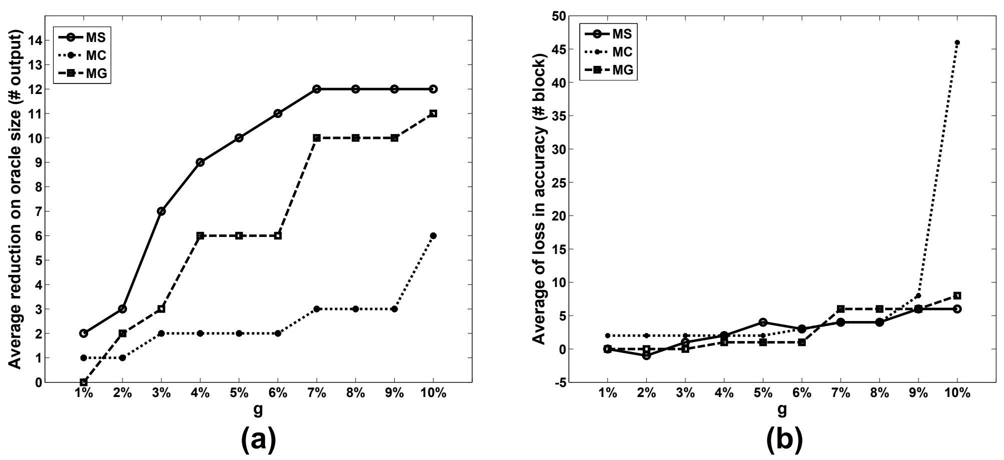


Figure 3.11. The impact of varying the value of g on the average reduction of oracle size (a) and the average loss in fault localization accuracy (b).

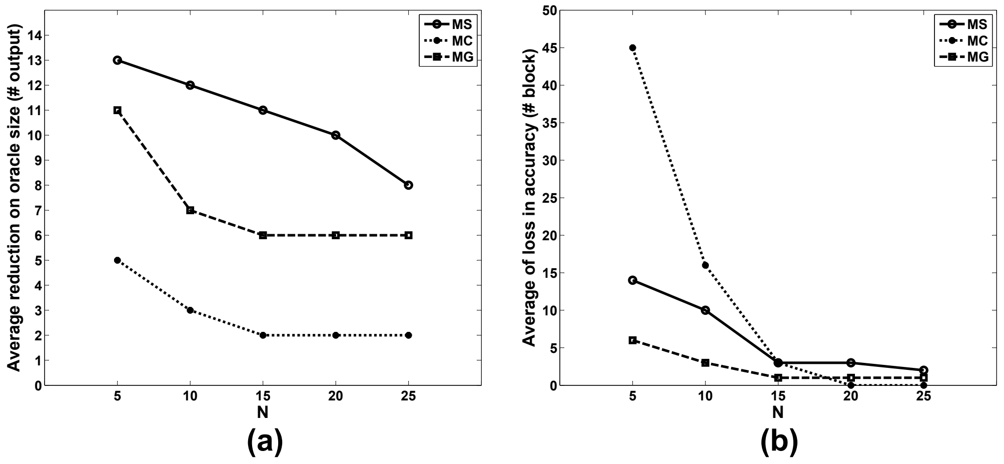


Figure 3.12. The impact of varying the value of N on the average reduction of oracle size (a) and the average loss in fault localization accuracy (b).

is used). This is because for larger g, the probability of finding rank groups with size larger than g decreases and *iSimFL*'s heuristic tends to extend test oracles less often (line 7 in Figure 3.5). Note that for *MS* and for two points $g = 1\%$ and $g = 5\%$, the accuracy slightly decreases when we increase g. This is because as we observed in **RQ3**, in some few cases by extending test oracles, accuracy may decrease. So although the relationship between g and oracle reduction is monotonic and fully predictable, i.e., oracle size decreases with increasing g, the relationship between g and accuracy loss is not always monotonic. However, as shown in Figure 3.11(b), in most cases by increasing g, accuracy loss either increases or stays the same, and only in two cases we may slightly gain accuracy by increasing g.

Similarly, when we fix g to 6% of the size of model (Figure 3.12) and increase N, *iSimFL* extends test oracles more (i.e., the difference between *iSimFL* required oracle size and size of $\mathcal{O}_2$ decreases), while the accuracy of ranking results increases (i.e., engineers on average have to inspect less blocks to find the fault compared to the number of blocks that they need to inspect when *SimFL* with $\mathcal{O}_2$ is used). Note that, when the value of N increases, *iSimFL* checks a larger number of most suspicious blocks for deciding whether the suspiciousness ranking is coarse or not. When the set of most suspicious blocks checked by *iSimFL* is larger, *iSimFL* is more likely to find a rank group with size > g (i.e., coarse ranking results), and hence, is more likely to decide that oracle extension is necessary. As a result, the average reduction on oracle size decreases. On the other hand, as shown in Figure 3.12(b), as the value of N increases, the accuracy of *iSimFL* gets closer to the accuracy of *SimFL* with $\mathcal{O}_2$, i.e., accuracy loss decreases. Note that the trend in Figure 3.12(b) happens to be monotonic, but as we discussed earlier, the changes in accuracy that are caused by changes in the oracle size are in general unpredictable.

In summary, the answer to **RQ5** is that changing the value of the parameters (i.e., N and g) used in *iSimFL* has a predictable impact on the oracle size required by *iSimFL*. By increasing g, oracle size decreases, and by increasing N, oracle size increases when compared to the size of the maximum oracle ($\mathcal{O}_2$). The accuracy loss is not always predictable when we change N and g. In a majority of cases, however, by increasing g, the accuracy loss increases, and by increasing N, the accuracy loss decreases when compared with the results obtained by *SimFL* with $\mathcal{O}_2$. Finally, based on Figure 3.11, we observe that when the value of g is between 4% to 6% of the model blocks, the average loss in fault localization accuracy is low (i.e., less than 5 blocks) for all the three models, while reduction in the test oracle size is relatively large (around 8 outputs on average for the three models). Based on Figure 3.12, we observe that the loss in accuracy is high when N is less than 15, suggesting that checking less than 15 most suspicious blocks may not be enough to assess the coarseness of ranking results and could lead to missing necessary oracle extensions, hence degrading *iSimFL*'s accuracy.

MC is the largest model but also has the smallest variation in oracle size from $\mathcal{O}_0$ to $\mathcal{O}_2$, i.e., there is less room for improvement compared to *MS* and *MG*. With the highest value of g and the smallest value of N, the heuristic leads to extending the oracle by only two more outputs, resulting in a larger loss of accuracy compared to *MS* and *MG*.

Based on the above results for three distinct models of different sizes, for the experiment whose results are reported in Figure 3.10, we picked optimal values for g and N, that is 6% and 15, respectively. When setting these parameters in practice, it does not make much sense for g to go higher than 10%, which is already a quite large rank group size. Further, g should not be below 4% of the model size since our results suggest that the reduction in oracle size will be limited. The parameter N is limited by how much time engineers have to inspect blocks. Our results suggest that, for our three subject studies, by setting N to be at least 15, i.e., less than 5% of the model size for our smallest model and less than 2% for our largest model, we are able to provide a reasonable prediction as to when the test oracle extensions are beneficial. Covering such small percentages of blocks is feasible in most practical contexts and situations.

3.5 Threats to Validity

Threats to the external validity relate to the generalizability of our findings. In this work, we evaluated the accuracy of our approach in localizing 95 faulty versions of three industrial Simulink models from the automotive domain. The industrial Simulink models that we analyzed are representative in terms of size and complexity among Simulink models developed at Delphi, and the seeded faults were realistic and were obtained from Delphi engineers. However, it is yet to be seen if our findings are generalizable to Simulink models from other domains.

Threats to the internal validity relate to the assumptions we made in our experiments. In particular, we evaluated our approach on faulty Simulink models where each faulty model contained one fault only. In practice, models may have multiple faults, and these faults may impact one another in unknown ways. However, a large bulk of existing research on applying statistical debugging to code is exclusively evaluated on programs seeded with single faults [Abreu et al., 2007, Arumuga Nainar and Liblit, 2010, Chen et al., 2002, Chilimbi et al., 2009, Cleve and Zeller, 2000, Cleve and Zeller, 2005, Jones and Harrold, 2005, Jones et al., 2002, Liblit et al., 2005, Liu et al., 2005, Lucia et al., 2010, Parnin and Orso, 2011, Renieris and Reiss, 2003, Santelices et al., 2009, Xie et al., 2013b]. Our approach is the first to apply statistical debugging to Simulink models, and no prior empirical results on Simulink fault localization exist. In our work, in order to be able to compare our findings with those reported in the literature, we decided to be consistent with the existing experiment settings and evaluate our approach on models seeded with single faults. Our work is a necessary basis before we can move forward to more complex evaluations involving models seeded with multiple faults. Further, our work opens up opportunities for more research on applying statistical debugging to Simulink models.

3.6 Conclusion and Future Work

We presented *SimFL*, a new fault localization approach for Simulink models by combining statistical debugging and dynamic model slicing. In our work, we generate finer grained spectra (i.e., one spectrum for each test case and each output) compared to the existing techniques where one test case yields a single spectrum. This allows us to apply statistical debugging to Simulink models where test suites are typically small due to the practical limits of embedded system development. We use backward static slicing and coverage reports to generate test execution slices. We then compute suspiciousness scores per block and per output using three different, well-known statistical ranking formulas and take the average of suspiciousness scores of each block over all outputs to obtain the final scores used for ranking. Our approach considers as many outputs as possible and necessary, potentially increasing test oracle cost. Hence, we propose an iterative fault localization algorithm (*iSimFL*) to help engineers determine when oracle extension is likely to increase accuracy. We applied *SimFL* to 95 faulty models generated based on three different Simulink models from the automotive industry. Our results show that *SimFL*'s accuracy in localizing faults in Simulink models is promising: on average, for example, using *SimFL* with *Tarantula*, the percentage of blocks inspected is at least 2.1% and at most 8.9% of the total model blocks. In contrast to fault localization for source code, we found that the accuracy of *Tarantula*, *Ochiai*, and D^2 in localizing fault in Simulink models are very similar. Further, we show that increasing the size of test suites, above what is common practice in embedded systems, does not significantly change *SimFL*'s accuracy. Hence, to improve accuracy, we extend test oracles using *iSimFL*, a method to iteratively refine them and augment their failure detection capability. We show that *iSimFL* significantly improves *SimFL*'s accuracy (i.e. on average, at least 1.3% and at most 4.4% of the total model blocks need to be inspected) by extending test oracles with only five outputs on average.

The performance of *iSimFL* depends on a stopping criterion heuristic, which is tunable via parameters N (the number of top most suspicious blocks inspected) and g (coarseness threshold). Our analysis shows that changing the value of N and g has a predictable impact on the test oracle size required by *iSimFL*. Further, for the majority of cases, the impact on the accuracy of *iSimFL* is also predictable. This is expected to facilitate the setting of such parameters. In this work, we relied on our experience and discussions with domain experts to determine the value of *iSimFL*'s parameters. Practical guidelines for choosing values for N and g require further studies and are left for future work.

Our results suggest that the three well-known statistical formulas (i.e., *Tarantula*, *Ochiai*, and D^2) yield similar accuracy in localizing faults for Simulink models. In future, we will investigate why different statistical formulas have similar impacts on fault localization accuracy. Moreover, we plan to extend *SimFL* to localize faults in Stateflow (state machine) models. In addition, we intend to perform user studies with engineers to better understand their information needs while debugging, so as to provide additional insights along with the block rankings.

Chapter 4

Localizing Multiple Faults in Simulink models

In the presence of multiple faults, faults may mask one another and different failures might be due to different faults. As statistical debugging implicitly assumes that all failures are caused by the same fault, the accuracy of statistical debugging for identifying faults often degrades when multiple faults exist in models. Thus, we have enhanced our fault localization approach to handle multiple faults in Simulink models.

Our approach is based on statistical debugging and iterative. At each iteration, our approach recommends a ranked list of most suspicious blocks. Engineers then inspect the recommended ranked-list to locate and resolve one fault. Further, engineers re-test the model to ensure that a particular fault is fixed, and to focus on localizing other faults including those that might have been masked or were not observable in the previous round of testing. Engineers apply our approach until no failures are observed.

We evaluate our approach on 240 multi-fault Simulink models obtained from three different industrial subjects. We have further adapted and implemented two baseline techniques for Simulink models: A traditional statistical debugging approach without clustering, and a state-of-the-art clustering-based statistical debugging that uses unsupervised learning [Jones et al., 2007]. We compared our decision tree-based approach with the two baseline techniques and our results show that our approach is able to significantly reduce the number of blocks that are required to be inspected to localize all faults compared with the two baselines and our approach is more robust than the baselines when applied to models containing larger numbers of faults.

This chapter highlights the following research contributions:

1. We propose a new iterative technique to localize multiple faults in Simulink models using a supervised learning technique (decision trees).
2. We evaluate our approach on 240 multi- fault models obtained from three different industrial subjects. We compare our approach with two baselines: (1) Statistical debugging without clustering, and (2) State-of-the-art clustering-based statistical debugging.

Organization. This chapter is organized as follows. Section 4.1 presents the motivation of our work in this chapter. Section 4.2 outlines our solution approach to localize multiple faults in Simulink models. Section 4.3 describes our experiments setup and evaluation results. Finally, Section 4.4 concludes the chapter.

4.1 Motivation

In this section, we motivate our approach to localizing multiple faults in Simulink models.

When models contain multiple faults, statistical debugging can be imprecise. This is because the multiple faults that exist in a model may impact one another in unknown ways causing the impact of some faults to be masked by others. This may result in faulty blocks to be ranked low in the rankings obtained by statistical debugging. To improve the results of statistical debugging in the presence of multiple faults, researchers have proposed to cluster failures in such a way that the failures that are caused by the same faults are put in the same cluster [Steimann and Frenkel, 2012, Liu and Han, 2006, Jones et al., 2007, Briand et al., 2007, Zheng et al., 2006, Jiang and Su, 2007].

Similar to the existing work [Steimann and Frenkel, 2012, Liu and Han, 2006, Jones et al., 2007, Briand et al., 2007, Zheng et al., 2006, Jiang and Su, 2007], we propose an approach based on failure clustering for identifying faults in Simulink models with multiple faults. Our approach is, however, different from the existing techniques in terms of the notion of failures, the input and the technique used for clustering, and in the way we use clustering results to localize faults. We explain each of these distinguishing factors below:

Notion of failures. As mentioned in Section 3.2.2, we associate a failure with the incorrect output of a test execution (i.e., a failing execution slice). Thus, in our work, clustering failures is equivalent to clustering failing execution slices. While in the existing techniques [Steimann and Frenkel, 2012, Liu and Han, 2006, Jones et al., 2007, Briand et al., 2007, Zheng et al., 2006], a failure corresponds to a failing test case and clustering failures is equivalent to clustering failing test cases without regard to the particular outputs at which failures are observed.

Input for clustering failures. Some existing techniques [Steimann and Frenkel, 2012, Jones et al., 2007, Zheng et al., 2006, Jiang and Su, 2007] take as input sequences of program elements executed by failing test cases, while other techniques [Liu and Han, 2006, Jones et al., 2007, Briand et al., 2007] use sequences of program elements executed by both passing and failing test cases. The inputs to our approach are sequences of blocks executed by each test case for each output, i.e., all test execution slices, as well as the test input data used to generate these slices. Our intuition is that failures are more likely to have been caused by the same fault, not only if they execute similar blocks, but also when they use similar test inputs.

Techniques for clustering failures. Most existing approaches [Liu and Han, 2006, Jones et al., 2007, Zheng et al., 2006, Jiang and Su, 2007] rely on *clustering techniques* (i.e., unsupervised learning techniques) where they group failures based on some similarity measure defined over the data that characterizes failures. Instead of relying on similarity measures, in this work, we use a *supervised learning technique* that can learn from failing and passing test executions to determine how to group the failures. Specifically, we use *Decision trees* [Olshen et al., 1984] (see Section 4.2.1). A decision tree is built by partitioning the set of test execution slices such that homogeneity is maximized across the resulting partitions, within certain constraints, in terms of passing and failing test execution slices. Decision trees also allow us to distinguish input data and execution trace characteristics that statistically determine failures.

Usage of clustering results. Similar to existing techniques [Steimann and Frenkel, 2012, Jones et al., 2007], we generate a single ranked list of most suspicious blocks per each cluster. However, instead of requiring engineers to inspect all ranked lists [Liu and Han, 2006, Steimann and Frenkel, 2012, Jones et al., 2007], our approach aims to select the most fault revealing ranked lists (i.e., those that rank faulty blocks higher), and requires engineers to inspect those selected ranked lists only (see Section 4.2.3). In our work, we assess the level of consistency of failing execution slices in clusters and we assume that the most consistent one will yield the best ranking.

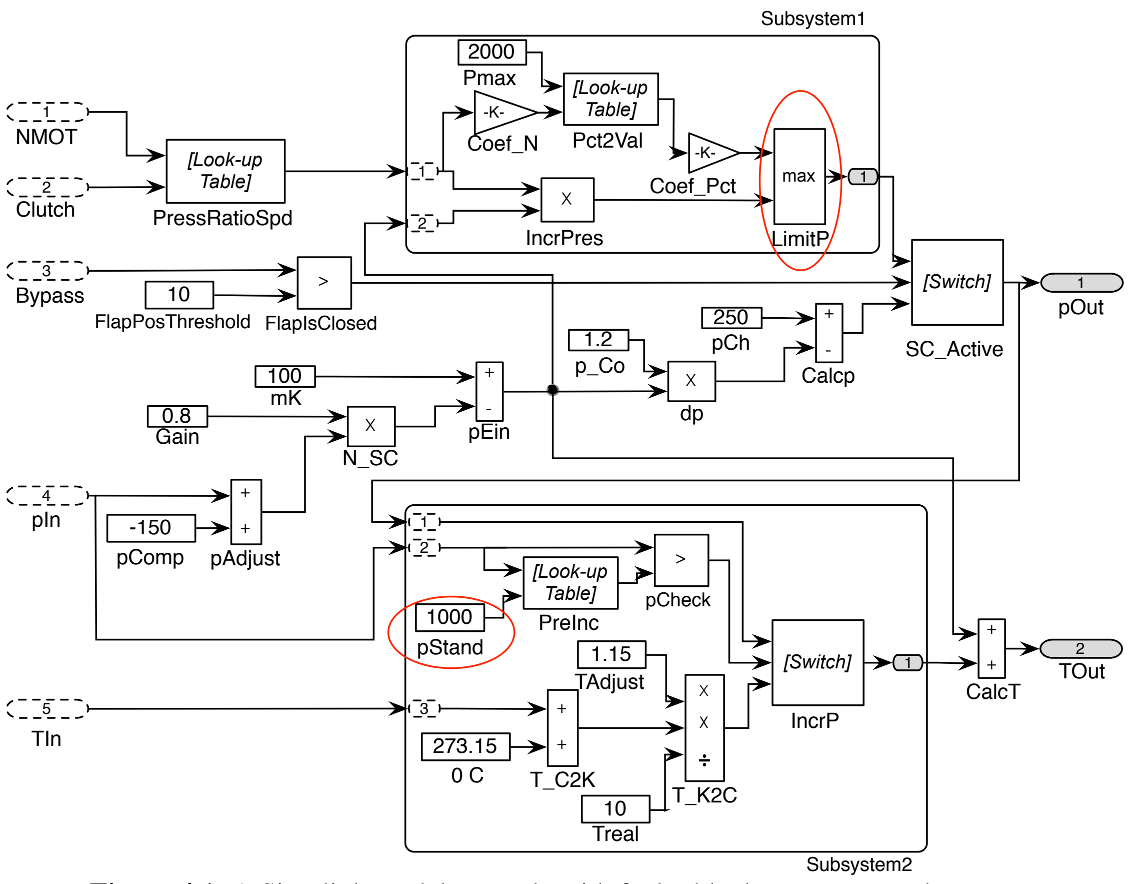


Figure 4.1. A Simulink model example with faulty blocks `LimitP` and `pStand`.

Motivating example. We illustrate the benefits of our statistical debugging approach that relies on clustering and is used in a *one-at-a-time* debugging process using the faulty model example in Figure 4.1 that contains two faults: in blocks `pStand` and `LimitP`. Table 4.1 shows that testing this model produces seven failures, three of which are caused by the fault in `LimitP` and the rest are due to the fault in `pStand` The block ranking computed based on *Tarantula* is shown in the left-most column of Table 4.1. In this ranking, the rank of `pStand` and `LimitP` are 12 and 14, respectively. Assuming engineers debug one fault at a time, they first locate the faulty block `pStand` by inspecting up to 12 blocks. After fixing this fault and re-applying the statistical debugging technique, engineers can locate the faulty block `LimitP` by inspecting at most three blocks. Thus, engineers need to inspect 15 blocks in total to locate both faults when they do not use clustering.

Block Name	tc_1		tc_2		tc_3		tc_4		tc_5		tc_6		Score	Rank (Min-Max)
	`pOut`	`TOut`	`pOut`	`TOut`	`pOut`	`TOut`	`pOut`	`TOut`	`pOut`	`TOut`	`pOut`	`TOut`		
SC_Active	✓	✓	✓		✓	✓	✓	✓	✓		✓	✓	0.52	15 - 17
***LimitP**	✓	✓	✓				✓	✓	✓		✓	✓	0.54	13 - 14
Coef_Pct	✓	✓					✓	✓	✓				0.32	27 - 30
Pct2Val	✓	✓					✓	✓	✓				0.32	27 - 30
Coef_N	✓	✓					✓	✓	✓				0.32	27 - 30
Pmax	✓	✓					✓	✓	✓				0.32	27 - 30
IncrPres			✓								✓	✓	1.00	1 - 1
PressRatioSpd	✓	✓	✓				✓	✓	✓		✓	✓	0.54	13 - 14
FlapIsClosed	✓	✓	✓		✓	✓	✓	✓	✓		✓	✓	0.52	15 - 17
FlapPosThreshold	✓	✓	✓		✓	✓	✓	✓	✓		✓	✓	0.52	15 - 17
Calcp					✓	✓							0.42	18 - 26
pCh					✓	✓							0.42	18 - 26
dp					✓	✓							0.42	18 - 26
p_Co					✓	✓							0.42	18 - 26
pEin		✓	✓	✓	✓	✓		✓		✓	✓	✓	0.59	2 - 12
mK		✓	✓	✓	✓	✓		✓		✓	✓	✓	0.59	2 - 12
N_SC		✓	✓	✓	✓	✓		✓		✓	✓	✓	0.59	2 - 12
Gain		✓	✓	✓	✓	✓		✓		✓	✓	✓	0.59	2 - 12
pAdjust		✓	✓	✓	✓	✓		✓		✓	✓	✓	0.59	2 - 12
pComp		✓	✓	✓	✓	✓		✓		✓	✓	✓	0.59	2 - 12
CalcT		✓		✓		✓		✓		✓		✓	0.59	2 - 12
IncrP		✓		✓		✓		✓		✓		✓	0.59	2 - 12
pCheck		✓		✓		✓		✓		✓		✓	0.59	2 - 12
PreInc		✓		✓		✓		✓		✓		✓	0.59	2 - 12
***pStand**		✓		✓		✓		✓		✓		✓	0.59	2 - 12
T_K2C				✓						✓			0.42	18 - 26
Tadjust				✓						✓			0.42	18 - 26
Treal				✓						✓			0.42	18 - 26
T_C2K				✓						✓			0.42	18 - 26
O C				✓						✓			0.42	18 - 26
Pass(P)/Fail(F)	P	P	F	P	P	F	P	F	F	F	F	F		

Table 4.1. Test execution slices and ranking results for Simulink model in Figure 4.1. * denotes the faulty blocks and ✓denotes the executed blocks.

When we use our decision tree-based clustering, we obtain two clusters as follows: *Cluster$_1$* consisting of the failing execution slices that are caused by the fault in `LimitP`; *Cluster$_2$* consisting of the failing execution slices that are caused by the fault in `pStand`. For each cluster, we generate a ranked list of the most suspicious blocks using Tarantula. We then select the most fault revealing ranked list to be inspected by engineers. For this example, our approach selects the ranked list generated from *Cluster$_1$* because it contains the most similar failing execution slices. By inspecting the ranked list from *Cluster$_1$*, engineers can find the faulty block `LimitP` by inspecting at most three blocks. We then re-apply our technique after fixing the fault at block `LimitP`. This time, our approach produces one cluster containing all the failing execution slices. Using the ranked list generated from this cluster, engineers can find the faulty block `pStand` by inspecting at most five blocks. Thus, engineers localize all faults by inspecting at most eight blocks which is significantly smaller than that of without clustering (i.e., 15 blocks).

4.2 Approach

In this section, we present our approach to localize multiple faults in Simulink models. Our approach (shown in Figure 4.2) takes as input a faulty Simulink model, a test suite, and test oracles to determine the pass/fail information for each model output and for each test execution. We describe the three steps of our approach in Sections 4.2.1 to 4.2.3.

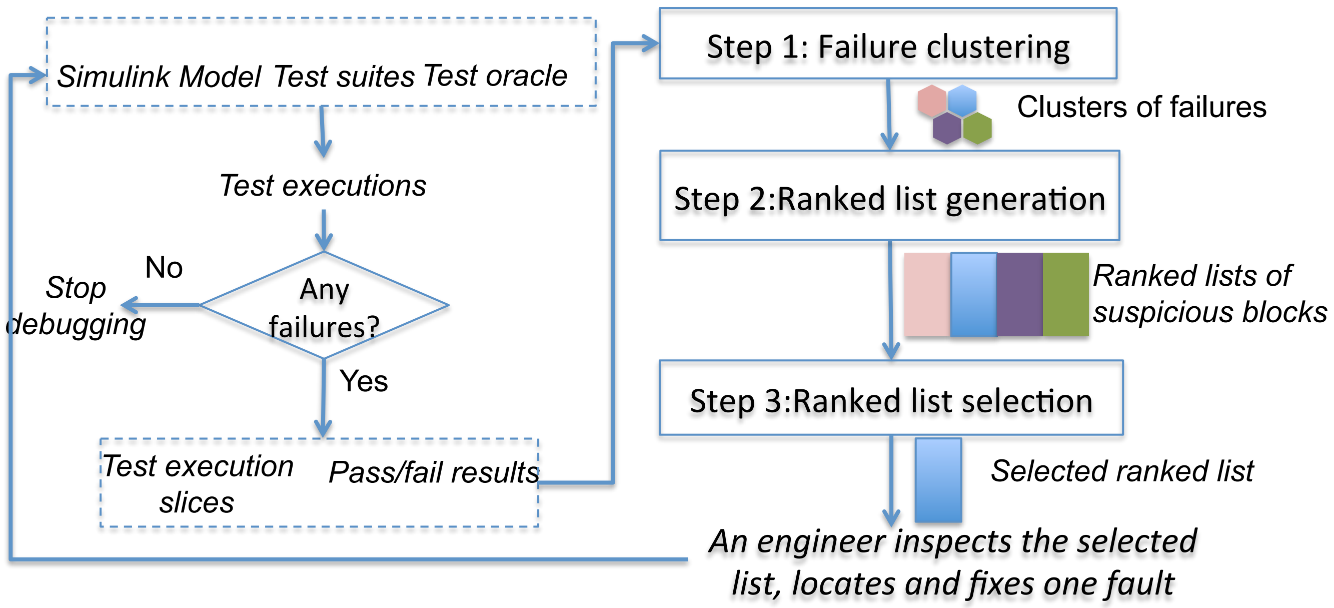

Simulink Model Test suites Test oracle
Test executions
Any failures?
No
Stop debugging
Yes
Test execution slices
Pass/fail results
Step 1: Failure clustering
Clusters of failures
Step 2:Ranked list generation
Ranked lists of suspicious blocks
Step 3:Ranked list selection
Selected ranked list
An engineer inspects the selected list, locates and fixes one fault

Figure 4.2. Overview of our approach to identify multiple faults in Simulink models.

4.2.1 Step 1. Failure Clustering

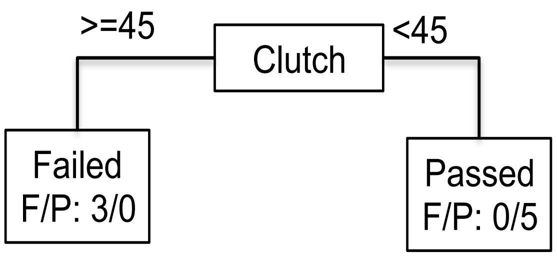
The goal of this step is to cluster failures such that the failures that are likely to have been caused by the same fault(s) are put in the same cluster. We cluster failures using *decision trees* [Olshen et al., 1984], a supervised learning technique. We apply the decision tree technique to a set of test execution slices. Each test execution slice contains the following information: (a) the blocks that are covered by the test execution slice; (b) the model input variables related to each test execution slice and the values of these model input variables. Each test execution is further labeled with passing (P) and failing (F) values. Consider our model example in Figure 4.1. Table 4.1 shows the blocks that are covered by each execution slice, and Table 4.2 shows the model input variables and values that are used by each execution slice. For example, the execution slice $(tc_3, TOut)$ covers 18 blocks (e.g., `pAdjust`), and is generated by two model inputs, i.e., `Bypass` and `pIn` with values 5 and 1500, respectively. Further, the execution slice $(tc_3, TOut)$ is labeled with F, indicating that the execution of tc_3 results in a failure at $TOut$.

Decision trees are composed of leaf nodes, which represent *partition*s, and non-leaf nodes, which represent *decision variables*. Given a set of failing and passing test execution slices, a decision tree is built by partitioning these slices in a stepwise manner with the aim of generating increasingly homogeneous partitions. A partition of test execution slices is fully homogeneous if the slices in that set are either all passing or all failing. The larger the gap between the number of failing and passing slices in a partition, the more homogeneous that partition is. A partition is labeled by `Failed` (respectively `Passed`) when the majority of the test execution slices in that partition are failing (respectively passing).

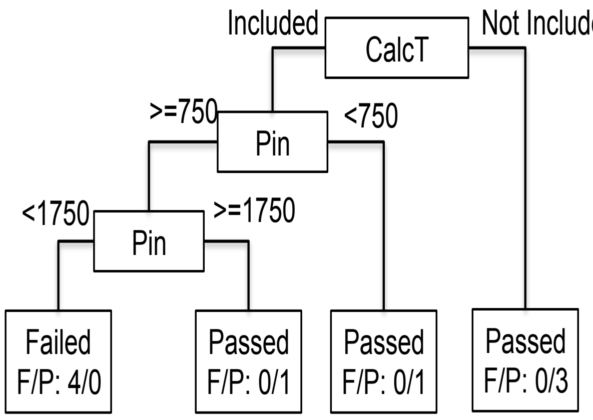
Decision variables in our decision trees either represent blocks or input variables. Given a decision variable (i.e., non-leaf node) labeled by block b, one branch (i.e., `included` branch) emanating from b leads to partitions (leaf nodes) containing slices all of which include b, and the other branch (i.e., `not included` branch) leads to partitions containing slices none of which include b. Given

Input block	*tc1*		*tc2*		*tc3*		*tc4*		*tc5*		*tc6*	
	`pOut`	`TOut`	`pOut`	`TOut`	`pOut`	`TOut`	`pOut`	`TOut`	`pOut`	`TOut`	`pOut`	`TOut`
NMOT	4500	4500	6000	-	-	-	4500	4500	3000	-	6000	6000
Clutch	40	40	50	-	-	-	40	40	50	-	50	50
ByPass	20	20	20	-	5	5	20	20	20	-	20	20
Pin	-	2000	500	500	1500	1500	-	1500	-	1000	1500	1500
Tin	-	-	-	10	-	-	-	-	-	10	-	-
Pass(P)/Fail(F)	P	P	F	P	P	F	P	F	F	F	F	F

Table 4.2. Model inputs and input values that are used to compute `pOut` and `Tout` for each test execution slice. Note that "-" denotes that the corresponding input value is not used to compute the corresponding output.



(4.3.1) A Decision tree for `pOut`.



(4.3.2) A Decision tree for `TOut`.

Figure 4.3. Decision trees generated for clustering failures at `pOut` and `TOut`.

a decision variable labeled by an input variable i, the two associated branches may be labeled as `included/not included` similar to the above, or alternatively, the branches may be labeled by conditional expressions on i (e.g., $i < 750$, $i \geq 750$). Formally, let P be a partition, and let $n_1 n_2 \ldots n_k P$ be the path to partition P from the root such that every $n_i (1 \leq i \leq k)$ is a non-leaf node representing a block or an input variable. As discussed above, every two consecutive nodes n_i and n_{i+1} are connected by a branch that is labeled by `included`, `not included`, or a conditional expression. The test execution slices in P consistently include (or exclude) the blocks and variables represented by nodes $n_i (1 \leq i \leq k)$, and further, they satisfy the conditions indicated by the path $n_1 n_2 \ldots n_k P$. Hence, the execution slices in P are likely to overlap, and are likely to have executed similar faulty blocks.

In our work, for each output that fails at least once, we build a decision tree that takes as input the failing test execution slices related to that output and all the passing test execution slices. Consider our faulty model example in Figure 4.1. We build the two decision trees in Figures 4.3.1 and Figures 4.3.2 that, respectively, relate to the failing outputs `pOut` and `TOut`. For example, the decision tree for `pOut` is built by using the three failing execution slices related to `pOut` (i.e., (tc_2,*pOut*), (tc_5,*pOut*)), (tc_6,*pOut*)) and all the passing execution slices (i.e., (*tc1*,*pOut*), (*tc1*,*TOut*),(*tc2*,*TOut*),(*tc3*,*pOut*), (*tc4*,*pOut*)). Note that `F/P` shown in Figure 4.3 indicates the number of failing and passing execution slices at each leaf node. Considering the decision tree for `pOut`, when the value of `Clutch` is greater than equal to 45, the test execution slices are likely to fail at output `pOut`. Otherwise, these slices are likely to be passing for `pOut`.

Note that decision trees do not require the number of partitions to be known a priori. Instead, to build such trees, we need to have precise criteria on when to terminate the partitioning, and on how decision variables should be selected to generate new partitions at each step. Given a partition, decision trees split the partition if the partition size is not smaller than a defined threshold (i.e., *minimum split* parameter) and if splitting the partition can reasonably reduce the miss-classification error [Olshen et al., 1984]. Further, decision trees rely on data homogeneity measures for selection of decision variables. In our work, we use the following termination and selection criteria to build our decision trees: We set the value of minimum split parameter to 50. This is because splitting partitions with size smaller than 50 would produce partitions with too few failing execution slices for statistical debugging to be able to distinguish faulty blocks from non-faulty ones. Moreover, we require that splitting a partition reduces the miss-classification error of decision trees by at least 1%. Finally, our decision trees use a well-known data homogeneity measure, namely *Gini Index* [Olshen et al., 1984].

Having built the decision trees, we create one cluster for each partition (leaf node) if that partition contains at least one failing execution slice. Each cluster contains only the failing execution slices (and not the passing slices) of their corresponding partitions. For example, using decision trees shown in Figure 4.3, we obtain two clusters: $Cluster_1 = \{(tc_2, pOut), (tc_5, pOut)), (tc_6, pOut)\}$ and $Cluster_2 = \{(tc3, TOut), (tc4, TOut), (tc5, TOut), (tc6, TOut)\}$.

4.2.2 Step 2. Ranked-List Generation

In this step, our approach generates a ranked list of most suspicious Simulink (atomic) blocks for each cluster produced in Step 1. Each ranked list indicates the blocks that are more likely to cause failures in the corresponding cluster.

To generate a ranked list for a cluster, we compute suspiciousness scores for Simulink blocks using the *Tarantula* formula [Jones and Harrold, 2005, Jones et al., 2002] (see equation in Section 3.2.3). Specifically, the *totalfailed* corresponds to the number of failing execution slices in a given cluster and the *totalpassed* is the total number of all passing execution slices. We then rank the blocks in descending order of their suspiciousness scores. As several blocks can obtain the same score, we assign min and max ranks for each block as described in Section 3.2.3.

Example. Using our example, we generate a ranked list for $Cluster_1$ and a ranked list for $Cluster_2$. To generate the ranked list for $Cluster_1$, we analyze failing execution slices in $Cluster_1$ and all the (five) passing execution slices. To generate the ranked list for $Cluster_2$, we analyze failing execution slices in $Cluster_2$ and all the (five) passing execution slices. In the ranked list obtained from $Cluster_1$, the faulty block *LimitP* obtains the highest rank (i.e., 3) with the score of 0.63, while in the ranked list obtained from $Cluster_2$, the faulty block *pStand* obtains the highest rank (i.e., 6) with the score of 0.71.

4.2.3 Step 3. Ranked-List Selection

In this step, we aim to select a ranked list that is more likely to yield the best fault localization results among the rankings generated in step 2. We introduce a ranked-list selection criterion, namely, *quality-of-cluster*. Prior to applying our selection criterion, we exclude the ranked lists obtained based on small clusters (i.e., the clusters that contain a few failing execution slices) from our selection pool.

This is because a small number of failing execution slices might not provide enough information to identify faults. In this work, a cluster is considered to be small if its size is smaller than 10.

The quality-of-cluster selection criterion aims to select the ranked lists that are generated from the most coherent cluster (i.e., clusters that contain similar failing execution slices). The rationale is that the more similar the failing execution slices, the more likely that they have executed the same faulty blocks. To measure the degree of similarity between slices inside a given cluster c, we define *intra-cluster distance* as the average of distances between the failing execution slices inside c. Let $D(S_i, S_j)$ be the distance between a pair of failing execution slices S_i and S_j. Given a cluster c, the quality of c (i.e., $QC(c)$) is the inverse of the intra-cluster distance of c denoted by $D_{Intra}(c)$, i.e., $QC(c)$=1/$D_{Intra}(c)$. We define $D_{Intra}(c)$ as follows:

$$D_{Intra}(c) = 2 \times \frac{\sum_{S_i, S_j \in c \wedge S_i \neq S_j} D(S_i, S_j)}{|c| \times (|c| - 1)} \tag{4.1}$$

The key to the quality-of-cluster criterion is the definition of distance $D(S_i, S_j)$ between pairs of failing execution slices. In our work, we provide two alternative definitions for $D(S_i, S_j)$ discussed as follows: (1) The intuition behind the first definition is that two failing execution slices are more similar (i.e., their pairwise distance $D(S_i, S_j)$ is small), if they execute similar sequences of blocks and are generated by similar model input variables with similar values. To capture this intuition, we associate to each failing execution slice S_i a vector S_i^v such that S_i^v has one element for each model block and one element for each model input variable. Specifically, the length of S_i^v is equal to the total of the number of model blocks and the number of input variables. Each element in vector S_i^v gets the following value: For each element of S_i^v related to a block b, we assign the element to one if S_i covers b, and otherwise, we assign zero. For each element of S_i^v related to an input value v, we assign v to the element and if that input is not covered by S_i, we assign *NaN* to the element.

(2) Based on the second definition, two failing execution slices are more similar (i.e., their pairwise distance $D(S_i, S_j)$ is small), if the sets of suspicious blocks that are produced based on those slices are more similar [Liu and Han, 2006, Jones et al., 2007]. To formalize this definition, we associate to each failing execution slice S_i a vector S_i^v such that S_i^v has one element for each model block (i.e., the size of S_i^v is equal to the number of model blocks). We then create a set of slices $\mathcal{S}$ containing S_i and all the passing execution slices, and use *Tarantula* to generate a ranking $\mathcal{R}$ based on the set $\mathcal{S}$. Then we obtain the top N elements from $\mathcal{R}$. In our work, we typically set N to be 10% of the model blocks. For each element of S_i^v related to a block b, we assign one to the element if b is among the top N elements obtained from $\mathcal{R}$. Otherwise, we assign zero to that element. Note that, this way of generating a ranking $\mathcal{R}$ for a failing execution slice S_i has been first proposed in [Jones et al., 2007] where the goal was to obtain a ranking for a failing test case.

We compute the pairwise distance $D(S_i, S_j)$ between failing execution slices based on each of the above two different definitions separately. Having computed vectors S_i^v (based on either of the two above definitions), for each failing execution slice S_i, we compute the distance $D(S_i, S_j)$ as the Euclidean distance between their corresponding vectors S_i^v and S_j^v. A small Euclidean distance indicates that two failing slices are similar. Note that when the first definition is used for S_i^v and S_j^v, the values of the vector elements representing input variables are equal to *NaN* or some values within the input variable ranges. Otherwise, the values of other elements of the vectors S_i^v and S_j^v are either one or zero by definition. In our Euclidean distance computation, instead of applying a subtraction operator to the elements representing input variables, we perform a matching that yields one if the values of these elements do not match and zero if their values match.

We denote the quality-of-cluster selection criterion by $QC_{Trace}(c)$ when the first definition above is used, and by $QC_{Rank}(c)$ when the second alternative definition is used. In either case, we select the ranked list that is obtained from clusters with the highest value of $QC_{Trace}(c)$ or $QC_{Rank}(c)$ (i.e., the clusters with smallest intra-cluster distance). If there are more than one cluster having the same quality, we randomly choose one of them.

Example. For *Cluster₁* and *Cluster₂* in our example, we have $QC_{Trace}(Cluster_1) = 0.48$ and $QC_{Trace}(Cluster_2) = 0.29$. Using the second definition of distance with $N = 5$, we have $QC_{Rank}(Cluster_1) = 1.19$ and $Cluster_2 = 0.49$. Hence, both QC_{Trace} and QC_{Rank} select the ranked list obtained from *Cluster₁* where the faulty block *LimitP* is ranked among the top three blocks.

4.3 Empirical Evaulation

In this section, we describe our research questions (Section 4.3.1), experiment settings (Section 4.3.2), evaluation metrics (Section 4.3.3), and experiment results (Section 4.3.4).

4.3.1 Research Questions

RQ1. [Fault Localization Accuracy] *Can our decision tree-based clustering approach help localizing faults by ranking the faulty blocks in the top most suspicious blocks? How does the fault localization ability of our approach compare with that of the non-clustering approach and the existing clustering approaches?* We investigate the accuracy of our approach in identifying faults in Simulink models with multiple faults. Specifically, we evaluate the maximum number of blocks inspected to identify faults at different debugging iterations. We compare our results with those obtained by two alternative debugging techniques for Simulink models with multiple faults: (1) Statistical debugging without using clustering (2) Statistical debugging combined with the pairwise clustering technique. The latter is a state-of-the-art clustering technique based on statistical debugging previously proposed for identifying multiple faults in source code [Jones et al., 2007]. We implemented and adapted this technique for Simulink models to use it as a baseline clustering technique for comparison with our work.

RQ2. [Fault Localization Cost] *Can our decision tree-based clustering approach significantly lower the cost of identifying all faults compared to the pairwise clustering and the non-clustering approaches?* We investigate the total cost of fault localization when our approach is used to identify several faults in Simulink models. We measure the total cost based on the total number of blocks that need to be inspected to make models fault-free. We then compare the total fault localization cost of our decision tree-based clustering approach with that of the non-clustering and the pairwise clustering approaches.

RQ3. [Robustness] *Does the fault localization ability of our approach remain robust when it is applied to models with an unknown (and potentially large) number of faults? How does our approach compare with the pairwise clustering and the non-clustering approaches in terms of robustness?* In

Model name	#Subsystem	#Blocks	#Links	#Inputs	#Outputs	# Fautly Versions
MS	37	646	596	12	8	80
MC	64	819	798	13	7	80
MGL	35	716	721	19	13	80

Table 4.3. Key information about industrial subjects.

order for our approach to be effective in localizing multiple faults, its fault localization ability should remain robust (i.e., show a graceful degradation) when the number of faults in the model grows. We study the changes in the fault localization ability of our approach when applied to models containing different numbers of faults and compare those changes with the changes in the fault localization ability of the non-clustering and the pairwise clustering approaches applied to the same models.

4.3.2 Experiment Settings

In this section, we describe the industrial subjects, test suites, and test oracles that are used for our experiments.

Industrial Subjects. We use three Simulink models developed by Delphi Automotive in our experiments. We refer to these three models as *MS*, *MC*, and *MGL*. Table 4.3 shows the number of subsystems, atomic blocks, links, and inputs and outputs of each model. Note that the models that we chose are representative in terms of size and complexity among the Simulink models developed at Delphi. Further, these models include about ten times more blocks than the publicly available Simulink models from the Mathworks model repository [MathWorks, a].

Fault Seeding. We requested a senior Delphi engineer to provide realistic faults for Simulink models based on his domain expertise and his years of experience in the automotive sector. We categorize the seeded faults into the following three groups: (1) *Wrong Function* such as using $>$ instead of $>=$. (2) *Wrong Connection* such as a wrong link between two blocks. (3) *Wrong Value* such as a wrong value in a constant block or a wrong threshold in a control block.

Based on the above set of faults, we seeded 19 faults into *MS*, 20 faults into *MC*, and 20 faults into *MGL* such that each fault is controlled by a switch allowing us to activate or deactivate each specific fault. Utilizing the fault activating/deactivating mechanism, we automatically created, for each model, 80 faulty versions containing different numbers of faults. Specifically, for each model, we created four sets of faulty versions of that model such that each set contains 20 faulty versions with *n* faults activated where *n* was set to two for the first set, to three for the second set, to four for the third set, and to five for the fourth set. We made sure to activate the faults in different parts of the models and of different types, and further, to cover all the originally seeded faults into each model. Overall, we created 240 faulty versions containing 840 faults in total.

Test suite and test oracle. We generated three test suites, each of which with 200 test cases for *MS*, *MC*, and *MGL* using Adaptive Random Testing [Chen et al., 2005]. Adaptive random testing is a black box and lightweight test generation strategy that distributes test cases evenly within valid input ranges, and thus, helps ensure diversity among test cases. Note that the size of the test suites was based on typical practice at Delphi given test budget constraints and the oracle costs. Further, we used the fault-free versions of our industrial subjects for test oracles.

Experiment design. To answer our research questions, we applied our approach (in a one-at-a-time debugging process) on our 240 faulty models. The number of debugging iterations for each faulty model is at most equal to the number of faults activated for that faulty model. This is because, at each iteration, we resolve all the faults located in the same rank as that of the top most ranked faulty block. For each faulty model and at each iteration, we run our approach outlined in Figure 4.2 by applying a test suite with 200 test cases to obtain test execution slices. We then subsequently apply the three steps in Figure 4.2 to generate a selected ranked list of suspicious blocks which is used by engineers to find one fault. We inspect the ranking manually to identify the first faulty block and we remove that fault by deactivating its corresponding switch. We then re-iterate the approach in Figure 4.2 until all faults are removed. We repeated the above experiment for the 240 faulty models twice: One time for the QC_{Trace} selection criterion, and the second time for the QC_{Rank} selection criterion. We denote our decision tree-based clustering approach that uses QC_{Trace}, QC_{Rank}, by *DT-QCTrace*, *DT-QCRank*, respectively.

As specified in the research questions, in our experiment, we consider two baseline techniques for comparison: A traditional statistical debugging technique without clustering (denoted by *NC*), and a statistical multi-fault debugging approach [Jones et al., 2007] that uses a *pairwise* clustering technique that we adapted to Simulink models. We repeated the above experiment for the *NC* and the *pairwise* approaches. Our implementation of the pairwise approach uses the setting used by Jones et al. [Jones et al., 2007] except that we consider the top 10% of the blocks to build the clusters as opposed to the top 20%. This is because in Chapter 3, we have shown that the top 10% of the Simulink blocks in a ranking are likely to contain most faults. As for the pairwise approach, since several clusters are generated, we use our two selection criteria discussed in Section 4.2.3 to select a ranking with the highest fault revealing ability. Specifically, we denote the pairwise approach that uses QC_{Trace}, QC_{Rank} by *PW-QCTrace*, *PW-QCRank*,respectively. In summary, we repeated our experiment 682, 679, 621, 673, and 659 times for *DT-QCTrace*, *DT-QCRank*, *PW-QCTrace*, *PW-QCRank*, and *NC*, respectively. Note that, at each iteration, we resolve all the faults located in the same rank as that of the top most ranked faulty block. Thus, different fault localization techniques require different numbers of iterations to resolve all the faults. We ran our experiment on a high performance computing platform with 2 clusters, 280 nodes, and 3904 cores. Our experiment were executed on different nodes of a cluster with Intel Xeon L5640@2.26GHz processor. The total computation time for our experiment (using a single node) is 15548 hours. Most of the experiment time is used to generate test execution slices. In total, we generated 1744000, 1503600, and 3000400 test execution slices for *MS*, *MC*, and *MGL*, respectively.

Decision tree settings. For building a decision tree, we use *Gini index* as the measure to determine which variables to be used for splitting. In order to control the size of our decision tree (i.e., in our work, the number of clusters of failure), a node in the tree can be split only if its size is larger than 50 i.e., the minimum split parameter equals to 50. We set this parameter to 50 in order to avoid nodes with very small size after splitting. Nodes that contain a small number of failing slices may not provide enough information to identify faults. Further, we set the value of the complexity parameter to 0.01 (i.e., default value) which is used to avoid splits that are not worthwhile i.e., when performing cross-validation on a decision tree, a split is not perfomed if the split does improve the accuracy of the decision tree in predicting failures and non-failure by 0.01.

4.3.3 Evaluation Metrics

Since we experiment with faulty models with multiple faults and fault localization is applied iteratively until models are fault-free, we provide two new metrics: *maximum rank of faulty blocks* and *fault localization cost*. In our work, at each iteration, we identify the faulty block that is ranked highest in the ranked list generated at that iteration. For each identified faulty block, the *maximum rank of faulty blocks* is the max rank of the rank group containing that faulty block. The *fault localization cost* is the total number of blocks that need to be inspected to localize all the faults in a given faulty model over all the iterations and making the model fault-free. Note that when several faults are in the same rank group in a ranked list, we assume that all of them are localized when engineers inspect that ranked list. Thus, the total iterations for obtaining a fault-free model can be smaller than the number of faults in that model. Note that the *fault localization cost* is an adaptation of the *absolute number of blocks inspected* metric used in the literature for *single-fault* localization in code [Parnin and Orso, 2011, Renieris and Reiss, 2003, Liu et al., 2005, Cleve and Zeller, 2005, Jones and Harrold, 2005, Lucia et al., 2014].

4.3.4 Experiment Results

4.3.4.1 RQ 1. Fault Localization Accuracy

To answer this question, we compute the number of faults and the proportion of faults that are ranked among the top blocks in some ranked list generated at some iteration by each of the *DT-QCTrace*, *DT-QCRank*, *PW-QCTrace*, *PW-QCRank*, and *NC* fault localization techniques. Figure 4.4 shows the number of faults that are ranked among the top blocks when these techniques are applied to our 240 faulty versions containing, in total, 840 faults. In this figure, the X-axis shows the number of top N ($N = \{10, 20, ..., 200\}$) blocks and the Y-axis shows the number of faults located among the top N blocks at some rank list produced at some fault localization iteration by each of the above five techniques. Based on Figure 4.4, when we use *DT-QCTrace* and *DT-QCRank*, 95 out of the total of 840 faults are ranked among the top 10 blocks in some ranked list at some iteration. In contrast, by using *NC*, *PW-QCTrace* and *PW-QCRank*, 23, 82, and 86 faults are ranked among the top 10 blocks at some iteration, respectively. In general, *DT-QCTrace* is able to rank more faults among the top ranked blocks compared to the other four techniques. After *DT-QCTrace*, *DT-QCRank* is the best. Further, both *DT-QCTrace* and *DT-QCRank* are better than *PW-QCTrace* and *PW-QCRank*. *NC* is worse than *PW-QCTrace* and *PW-QCRank* when we are interested in faults ranked among the top 50 blocks.

Figures 4.5 to Figures 4.8 show the proportion of faults that are ranked among the top blocks when our five fault localization techniques are, respectively, applied to the 120 faults seeded into our two-fault models, the 180 faults seeded into our three-fault models, the 240 faults seeded into our four-fault models, and the 300 faults seeded into our five-fault models, respectively. Note in Figures 4.5 to Figures 4.8, the X-axis is the same as the X-axis in Figure 4.4, but the Y-axis shows the proportion (instead of the absolute number) of faults ranked high, because the numbers of faults seeded into two-fault to five-fault models are different form one another. Based on Figure 4.5, using *DT-QCTrace*, 61 out of the 120 faults (i.e., more than 50% of the faults) seeded into the two-fault models are ranked among the top 50 blocks. In contrast, using *DT-QCRank*, *NC*, *PW-QCTrace*, and *PW-QCRank*, the 61 faults are ranked among the top most 70, 60, 90, and 90 blocks, respectively.

In general, the results in Figures 4.5 to Figures 4.8 show that *DT-QCTrace* and *DT-QCRank* always

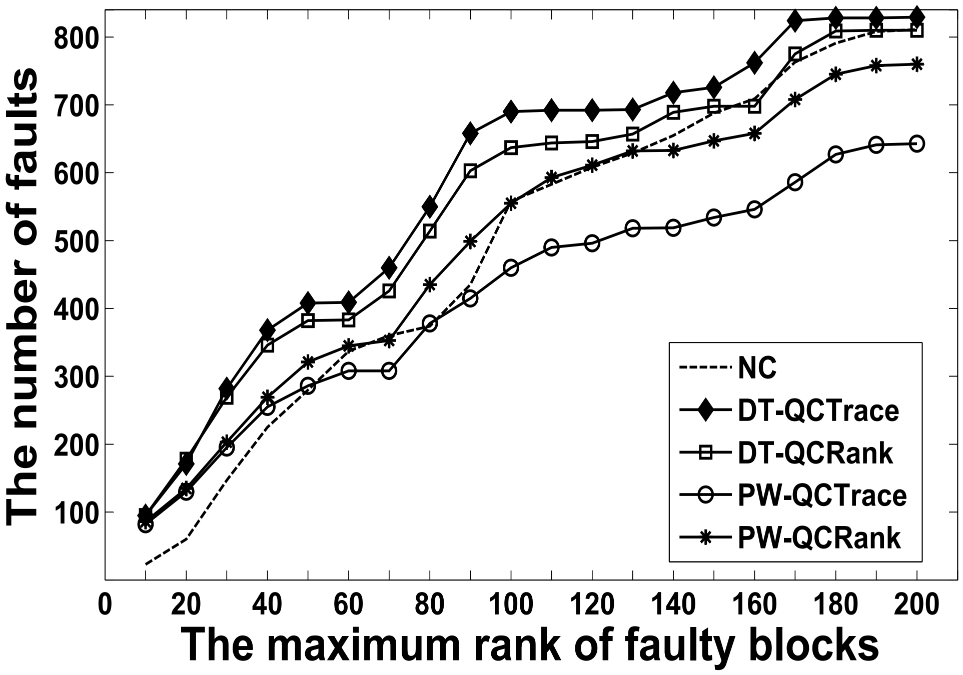


Figure 4.4. The number of faults vs. the maximum rank of faulty blocks for all the 840 faults.

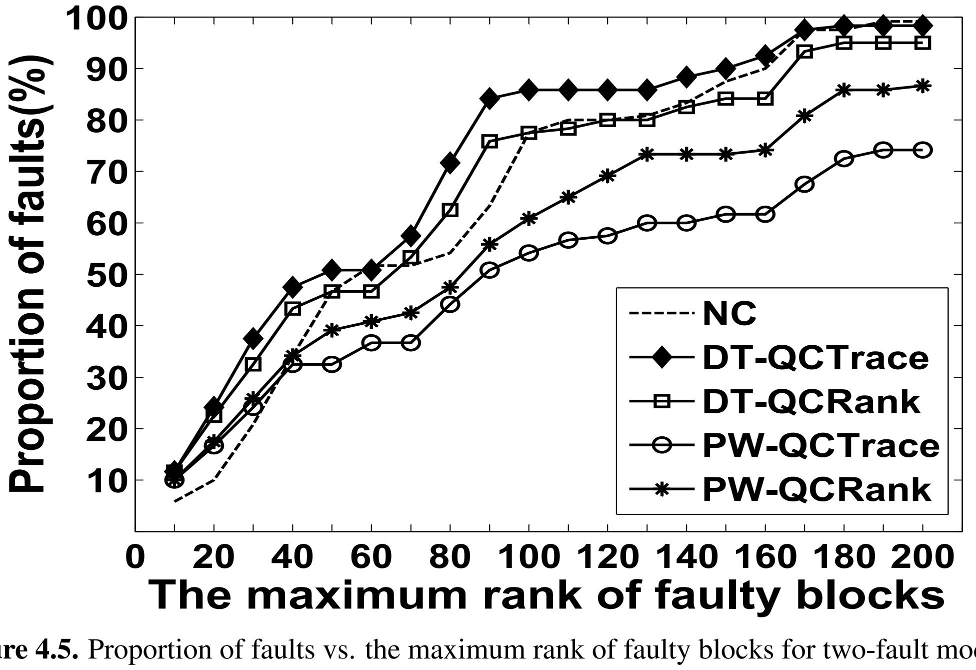


Figure 4.5. Proportion of faults vs. the maximum rank of faulty blocks for two-fault models.

perform better than *PW-QCTrace* and *PW-QCRank*. Further, *DT-QCTrace* always performs better than *NC* for three-fault to five-fault models, and also for two-fault models when we consider the faults that are ranked among the top 60 blocks. *DT-QCRank* always performs better than *NC* when we consider the faults that are ranked among the top 50 blocks. Note that it is expected for *NC* to eventually converge to the same performance as that of our clustering technique (*DT-QCTrace* and *DT-QCRank*) when the number of faults in our models are small (e.g., two-fault models). This is because faults are less likely to mask one another, and hence, the rankings generated by *NC* are less impacted when the number of faults are small (e.g. two faults).

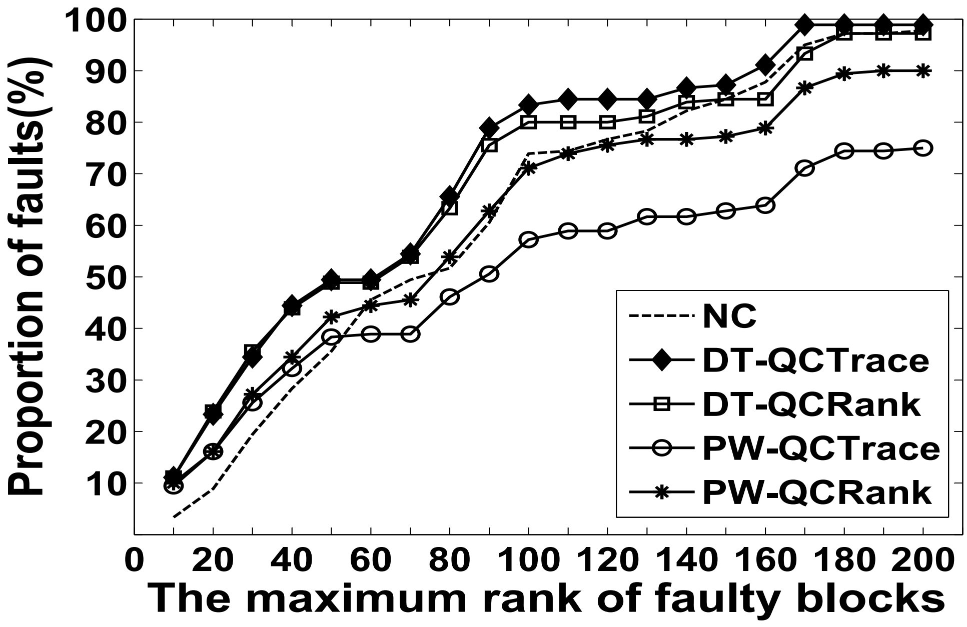


Figure 4.6. Proportion of faults vs. the maximum rank of faulty blocks for three-fault models.

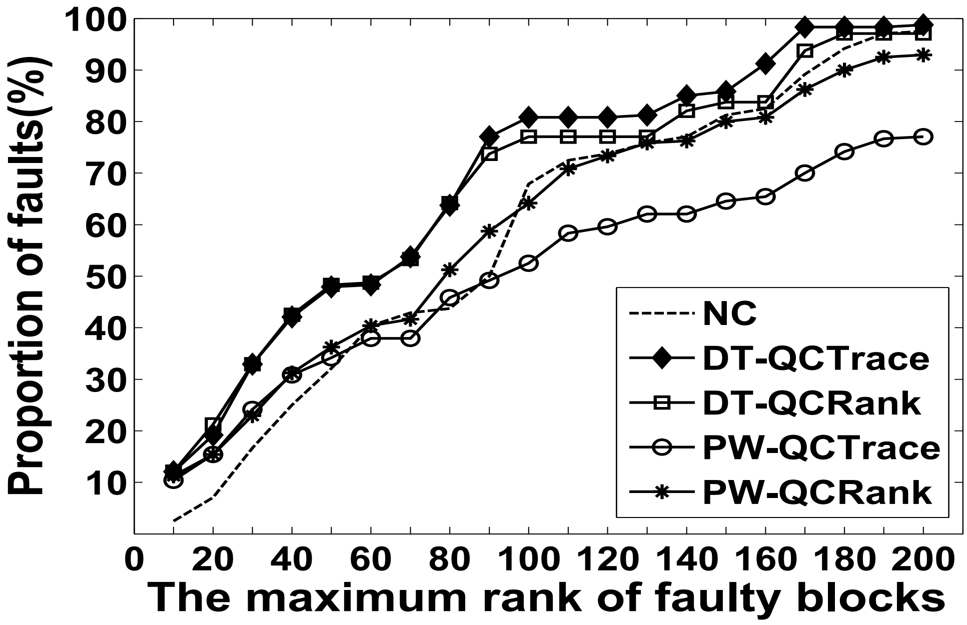


Figure 4.7. Proportion of faults vs. the maximum rank of faulty blocks for four-fault models.

In summary, the answer to **RQ1** is that our decision tree-based clustering approach is able to rank the faulty blocks among the top most suspicious blocks. Specifically, our techniques (i.e., *DT-QCTrace* and *DT-QCRank*) always outperform the statistical debugging with pairwise clustering (i.e., a state-of-the-art fault localization clustering technique). Further, *DT-QCTrace* always performs better than *NC* (i.e., the baseline non-clustering technique) except when the number of faults in models are small (i.e., two). For two-fault models, *DT-QCTrace* always perform better that *NC* when we consider the faults that are ranked among the top 60 blocks.

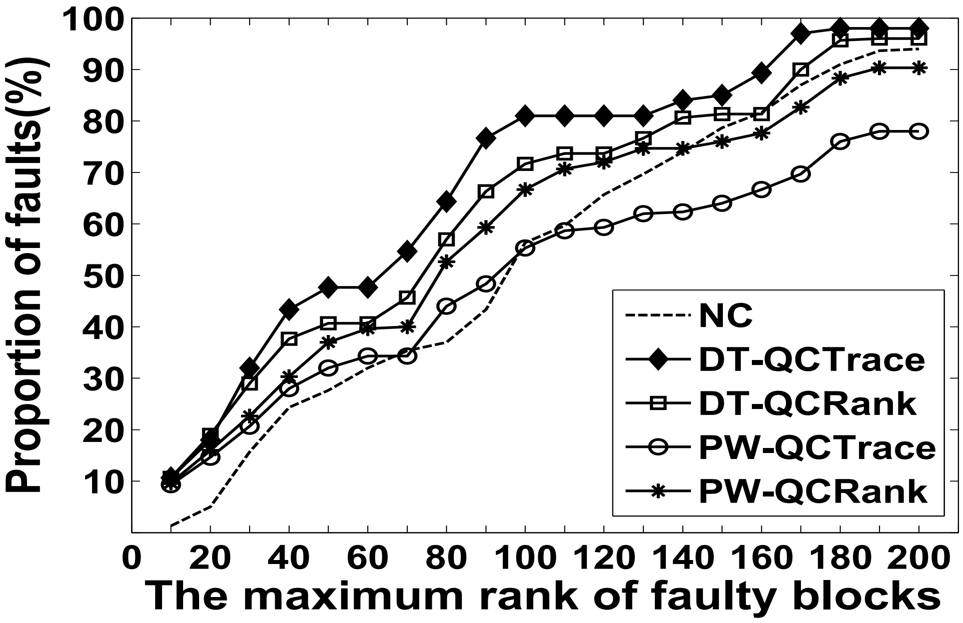


Figure 4.8. Proportion of faults vs. the maximum rank of faulty blocks for five-fault models.

4.3.4.2 RQ2. Fault Localization Cost

To answer this question, we compute the fault localization cost values for all the 240 faulty models and for each of the *DT-QCTrace*, *DT-QCRank*, *PW-QCTrace*, *PW-QCRank*, and *NC* fault localization techniques. Figures 4.9 to Figures 4.12 show the distributions of the fault localization cost values for our two-fault to five-fault models, respectively. Specifically, each box-plot consists of 60 points corresponding to the 60 faulty versions in each of the two-fault to the five-fault model groups. In each of these figures, the X-axis shows the five fault localization techniques, and the Y-axis shows the fault localization cost.

To statistically compare the fault revealing ability of different fault localization techniques, we performed the *non-parametric pairwise Wilcoxon signed-rank tests* [Wilcoxon, 1945], and calculated the effect size using *Vargha and Delaney A test* [Vargha and Delaney, 2000]. The A_{12} was labeled "small" for $0.56 \leq d < 0.64$, "medium" for $0.64 \leq d < 0.71$, and "high" for $d \geq 0.71$ [Vargha and Delaney, 2000].

Based on the statistical test results, for two-fault to five-fault models, the fault localization cost of our decision tree-based approaches (*DT-QCTrace* and *DT-QCRank*) is always significantly lower (better) than that of the other three techniques (*NC*, *PW-QCTrace*, and *PW-QCRank*) (p-values < 0.01). Further, the fault localization cost of *DT-QCTrace* is always significantly lower (better) than that of *DT-QCRank*. The effect size, when comparing *DT-QCTrace* and *NC*, is "small" for two-fault, three-fault and four-fault models, and "large" for five-fault models. In addition, when comparing *DT-QCTrace* with *PW-QCTrace*, is "large" for two-fault, four-fault, and five-fault models, and "medium" for three-fault models. When comparing *DT-QCTrace* with *PW-QCRank*, "small" for four-fault models, and "medium" for two-fault, three-fault and five-fault models.

In summary, the answer to **RQ2** is that our decision tree-based techniques significantly improve the fault localization cost compared to *NC*, *PW-QCTrace*, and *PW-QCRank*. Further, on average, *DT-QCTrace* reduces the fault localization cost by 59 blocks (25%) compared to *NC*, and by 62 blocks

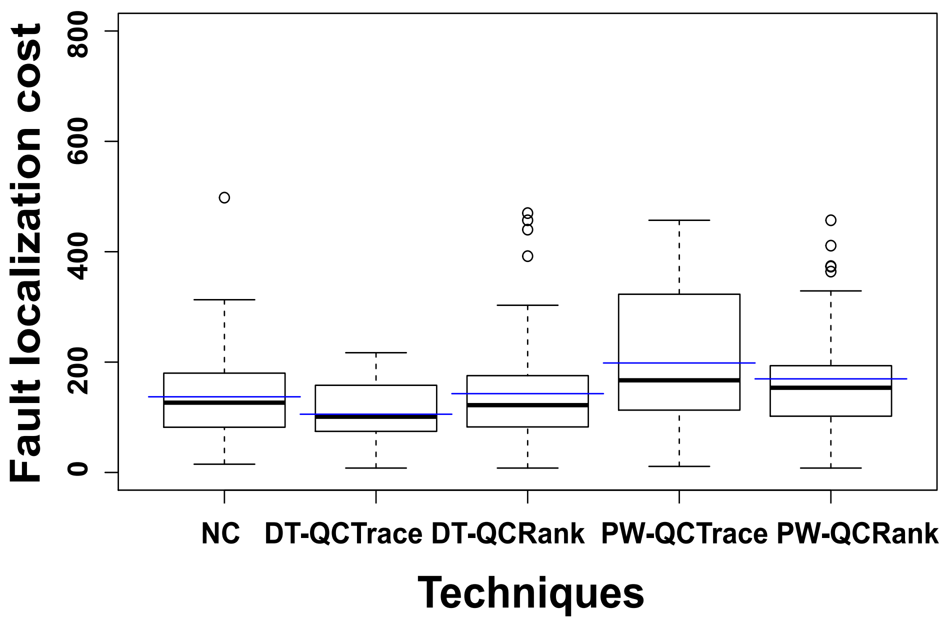


Figure 4.9. Distributions of fault localization cost for two-fault models.

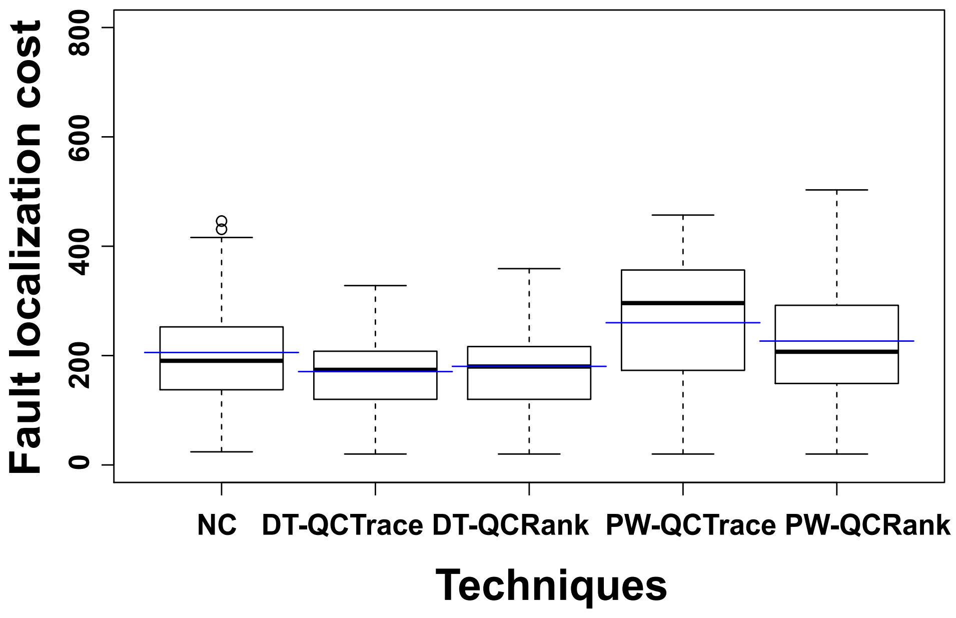


Figure 4.10. Distributions of fault localization cost for three-fault models.

(26%) compared to *PW-QCRank*.

4.3.4.3 RQ3. Robustness

For this question, we consider *DT-QCTrace*, *PW-QCRank* and *NC* because based on our results in **RQ1** and **RQ2**, *DT-QCRank*, *PW-QCTrace* underperform *DT-QCTrace* and *PW-QCRank*, respectively. To answer this question, we evaluated the changes in the proportion of faults that are ranked among the top blocks as we vary the number of faults seeded in the underlying faulty models. Figures 4.13 to Figures 4.15 show the results for *DT-QCTrace*, *PW-QCRank*, and *NC*, respectively. In each figure, we show how the performance of each of these three techniques is impacted when that

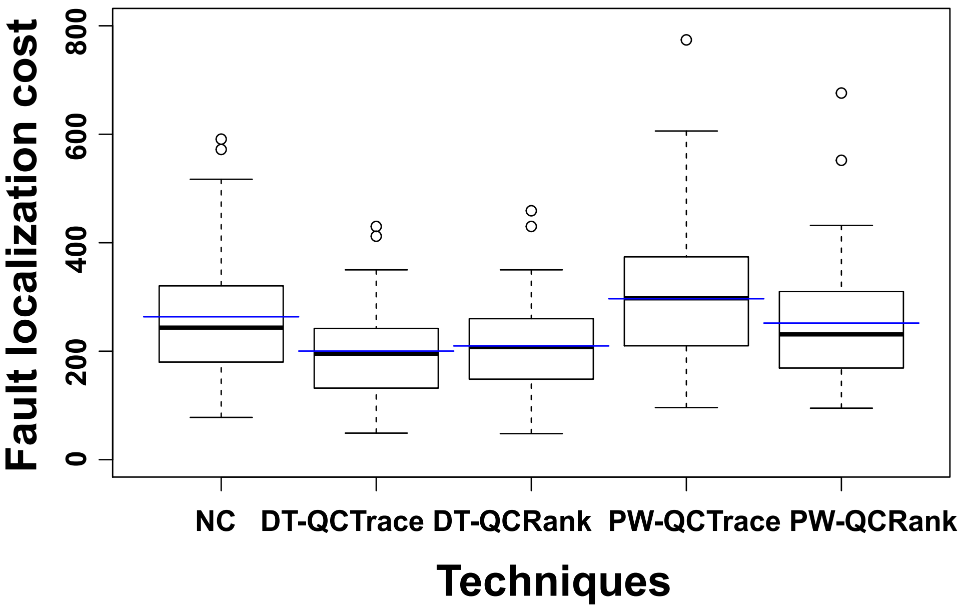


Figure 4.11. Distributions of fault localization cost for four-fault models.

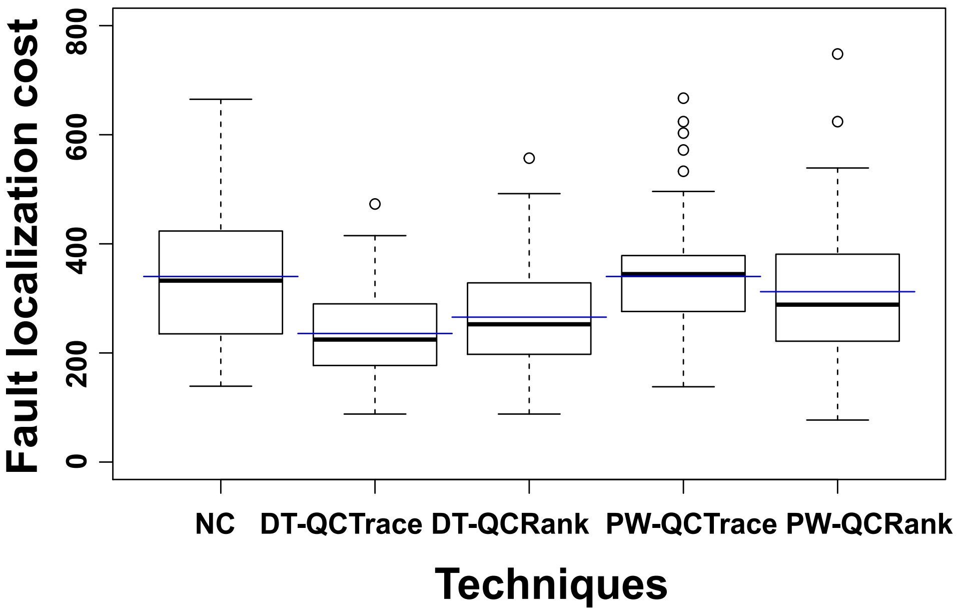


Figure 4.12. Distributions of fault localization cost for five-fault models.

technique is applied to the two-fault, the three-fault, the four-fault, and the five-fault models separately.

The data in Figures 4.13 to Figures 4.15 were already shown in Figures 4.5 to Figures 4.8 where we showed that *DT-QCTrace* outperforms other techniques. Figures 4.13 to Figures 4.15, however, compare the robustness of these techniques as the number of faults changes. As this figure shows, *DT-QCTrace* is the most robust technique since its performance changes the least as the number of faults increases from two to five. The maximum deviation for *DT-QCTrace* is 7.9% and the average deviation is 3.7%. *PW-QCRank* is less robust than *DT-QCTrace* but more robust than *NC* with maximum and average deviations of 10.3% and 5.5%, respectively. Finally, *NC* is the least robust

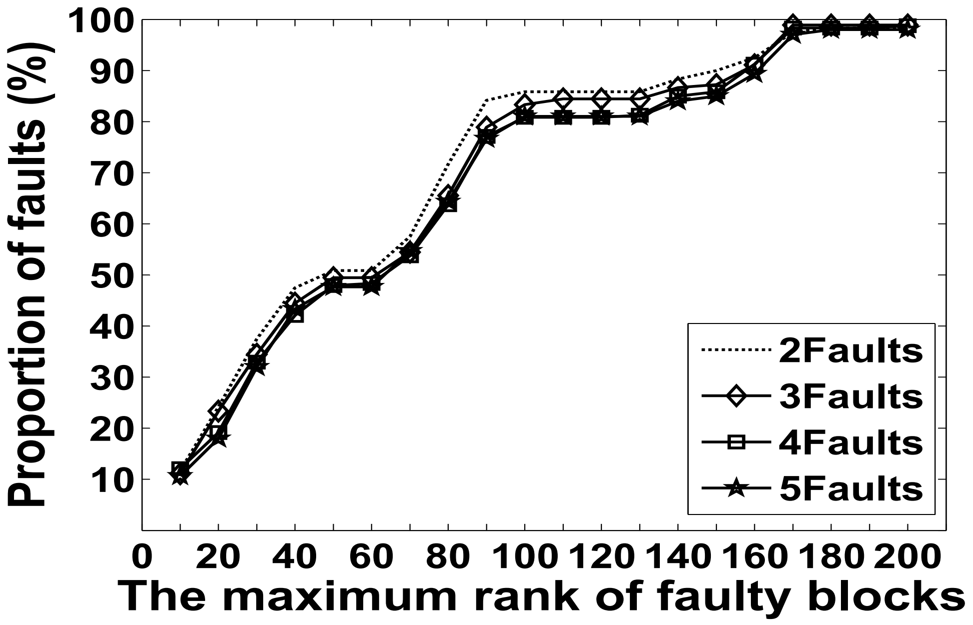


Figure 4.13. Proportions of faults (y-axis) among the top-N ranks (x-axis) obtained by *DT-QCTrace*.

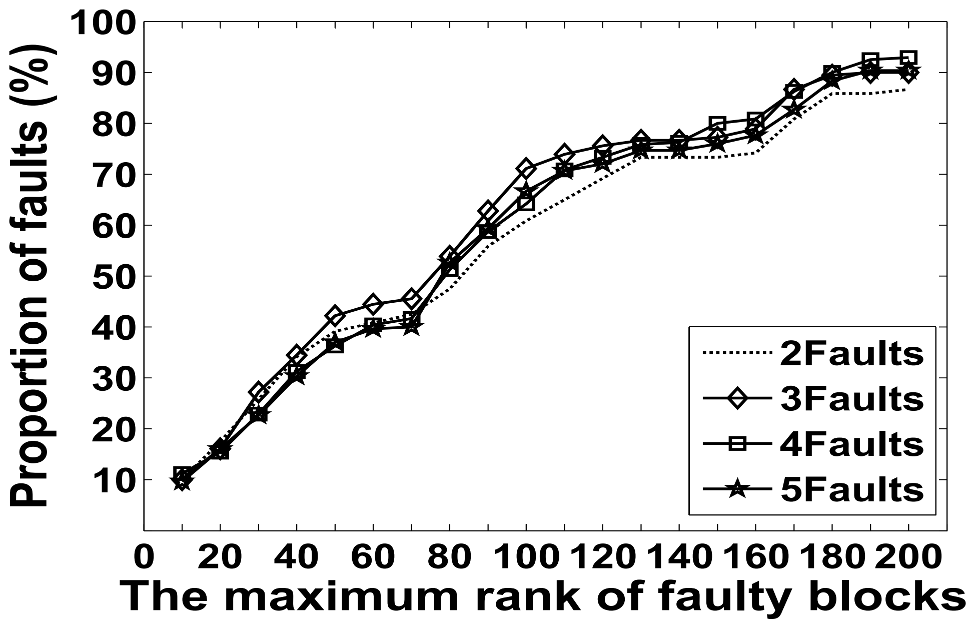


Figure 4.14. Proportions of faults (y-axis) among the top-N ranks (x-axis) obtained by *PW-QCRank*.

technique with maximum and average deviations of 21.2% and 11.9%, respectively.

In summary, compared to *PW-QCRank* and *NC*, the fault localization ability of *DT-QCTrace* is more robust as the number of faults in models increases from two to five faults. The least robust technique among these three is *NC*.

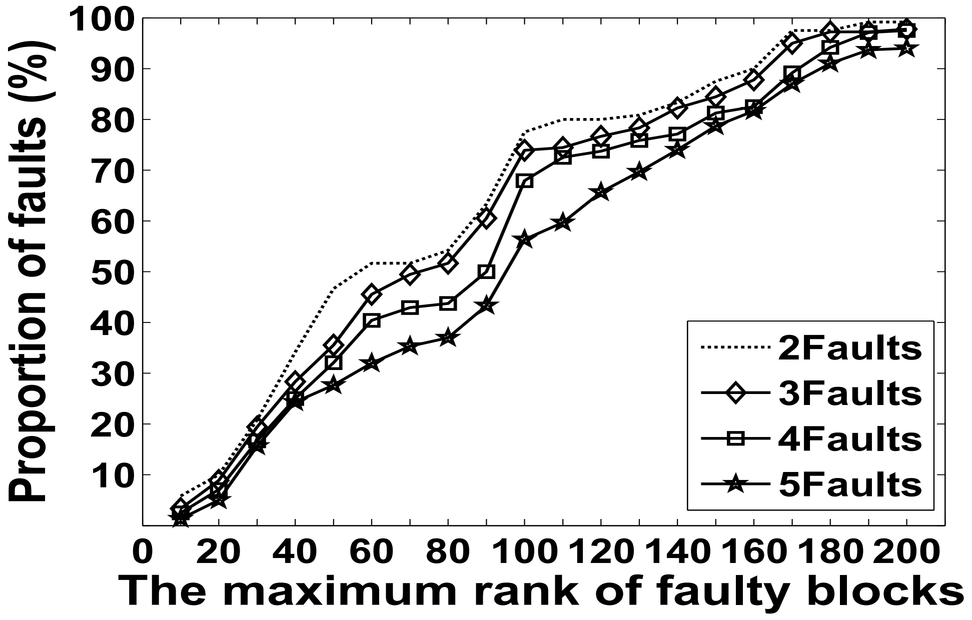


Figure 4.15. Proportions of faults (y-axis) among the top-N ranks (x-axis) obtained by *NC*.

4.4 Conclusion

In this chapter, we propose an approach to localize multiple faults in Simulink models. Our approach clusters failures (i.e., failing execution slices) that are likely to have been caused by the same fault(s) by using decision trees. Decision trees group together failures that satisfy similar (logical) conditions on model blocks and test inputs. For each cluster, our approach generates a ranked list of most suspicious blocks. We then select a ranked list that is the most likely to have a faulty block ranked high. Engineers then inspect this list to find at least one fault, fix the fault, and re-test the models. Our approach iterates until no failures are observed. We have evaluated our approach on 240 multi-fault models obtained from three different industrial subjects. Our experiment results show that our approach, on average, reduces the number of blocks inspected to localize all faults by 59 blocks (25%) compared to statistical debugging without clustering and by 62 blocks (26%) compared to a state-of-the-art pairwise clustering approach. These reductions are statistically significant with p-values less than 0.01. Furthermore, our approach exhibits less performance degradation than the baselines when we increase the number of faults in the underlying models. In future, we plan to provide effective visualization mechanisms to help engineers debug Simulink models.

In our chapter, we studied these fault types: wrong function, wrong value, and wrong connection. In future, we plan to consider other fault types, e.g., missing blocks or missing connections. Further, we plan to localize multiple faults in Stateflow [MathWorks, b] (state machine) models.

Chapter 5

Test Suite Generation for Improving Fault Localization

In this chapter, we focus on improving fault localization of Simulink models by generating test cases. We identify three similar approaches in the literature where the accuracy of statistical debugging is improved by generating additional test cases that help increase diversity in the underlying test suite used for debugging [Jiang et al., 2009, Campos et al., 2013, Baudry et al., 2006]. We adapt the test objectives used in these earlier papers to Simulink models and use these test objectives to develop a search-based test generation algorithm, which builds on the whole test suite generation algorithm [Fraser and Arcuri, 2013], to extend an existing test suite with a small number of test cases. Given the heuristic nature of statistical debugging, adding test cases may not necessarily improve fault localization accuracy. Hence, we use the following two-step strategy to stop test generation when it is unlikely to be worthwhile: First, we identify Simulink super blocks through static analysis of Simulink models. Statistical debugging, by definition, always ranks the blocks inside a super block together in the same rank group. Thus, when elements in a rank group are all from a super block, the rank group cannot be further refined through statistical debugging, and hence, test generation is not beneficial. Second, we develop a prediction model based on supervised learning techniques, specifically decision trees [Olshen et al., 1984], using historical data obtained from previous applications of statistical debugging. Our prediction model effectively learns rules that relate improvements in fault localization accuracies to changes in statistical rankings obtained before and after adding test cases.

We have evaluated our approach on 60 faulty versions obtained from three different industrial subjects. Our experiment results show that the three test objectives are able to significantly improve the accuracy of fault localization for small test suite sizes. Our approach is able to reduce the average number of blocks need to inspect to find a fault from 76 to 43 blocks (i.e. 43.4% reduction) by adding only 25 test cases. Moreover, when engineers use our prediction model, we are able to maintain almost the same fault localization accuracy while reducing the average number of newly generated test cases by more than half (i.e., 52% to 56% fewer test cases).

This chapter highlights the following research contributions:

1. We develop a search-based testing technique for Simulink models that uses the existing alternative test objectives [Jiang et al., 2009], [Campos et al., 2013], [Baudry et al., 2006], to generate small and diverse test suites that can help improve fault localization accuracy.

2. We develop a strategy to stop test generation when test generation is unlikely to improve fault localization. Our strategy builds on static analysis of Simulink models and prediction models built based on supervised learning.
3. We have evaluated our approach using three industrial subjects.

Organization. This chapter is organized as follows. Section 5.1 provides the motivation of our approach in this chapter. Section 5.2 presents our solution approach to generate test suite for improving fault localization for Simulink models. The results of our evaluation of the proposed approaches are presented in Section 5.3. Finally, Section 5.4 concludes the chapter.

5.1 Motivation

Statistical debugging is a lightweight and well-studied debugging technique [Abreu et al., 2007, Jones et al., 2002, Liblit et al., 2005, Liu et al., 2005, Renieris and Reiss, 2003, Santelices et al., 2009, Wong et al., 2008, Wong et al., 2014]. Statistical debugging localizes faults by ranking program elements based on their suspiciousness scores. These scores capture faultiness likelihood for each element and are computed based on statistical formulas applied to sequences of executed program elements (i.e., spectra) obtained from testing. Developers use such ranked program elements to localize faults in their code.

In Chapter 3, we extended statistical debugging to Simulink models and evaluated the effectiveness of statistical debugging to localize faults in Simulink models. Our approach builds on a combination of statistical debugging and dynamic slicing of Simulink models. We showed that the accuracy of our approach, when applied to Simulink models from the automotive industry, is comparable to the accuracy of statistical debugging applied to source code (discussed in Chapter 3). We further extended our approach to handle fault localization for Simulink models with multiple faults (discussed in Chapter 4).

Since statistical debugging is essentially heuristic, despite various research advancements, it still remains largely unpredictable [Campos et al., 2013]. In practice, it is likely that several elements have the same suspiciousness score as that of the faulty, and hence, be assigned the same rank. Engineers will then need to inspect all the elements in the same rank group to identify the faulty element. Given the way statistical debugging works, if every test case in the test suite used for debugging executes either both or neither of a pair of elements, then those elements will have the same suspiciousness scores (i.e., they will be put in the same rank group). One promising strategy to improve precision of statistical debugging is to use an existing ranking to generate additional test cases that help *refine* the ranking by reducing the size of rank groups in the ranking [Baudry et al., 2006, Campos et al., 2013, Rö*β*ler et al., 2012, Artzi et al., 2010].

In situations where test oracles are developed manually or when running test cases is expensive, adding test cases is not a zero-cost activity. Therefore, an important question, which is less studied in the literature, is how we can refine statistical rankings by generating a *small* number of additional test cases? In this chapter, we aim to answer this question for fault localization of Simulink models. While our approach is not particularly tied to any modeling or programming language, we apply our work to Simulink since, in some domains (e.g., automotive), it is expensive to execute Simulink models and to characterize their expected behaviour [Zander et al., 2011, Matinnejad et al., 2016]. This is because Simulink models include computationally expensive physical models [Ben Abdessalem et al., 2016], and their outputs are complex continuous signals [Matinnejad et al., 2016]. We identify three alternative test objectives that aim to generate test cases exercising diverse parts of the underlying code and adapt these objectives to Simulink models [Jiang et al., 2009, Campos et al., 2013, Baudry et al., 2006]. We use these objectives to develop a search-based test generation algorithm, which builds on the whole test suite generation algorithm [Fraser and Arcuri, 2013], to extend an existing test suite with a small number of test cases. Given the heuristic nature of statistical debugging, adding test cases may not necessarily improve fault localization accuracy. Hence, we use the following two-step strategy to stop test generation when it is unlikely to be worthwhile: First, we identify Simulink *super blocks* through static analysis of Simulink models. Given a Simulink model M, a super block is a set B of blocks of M such that, for any test case tc, tc executes either *all* or *none* of the blocks in B. That is, there is no test case that executes a subset (and not all) of the blocks in a super block. Statistical debugging, by definition, always ranks the blocks inside a super block together in the same rank group. Thus, when elements in a rank group are all from a super block, the rank group cannot be further refined through statistical debugging, and hence, test generation is not beneficial. Second, we develop a prediction model based on supervised learning techniques, specifically decision trees [Olshen et al., 1984] using historical data obtained from previous applications of statistical debugging. Our prediction model effectively learns rules that relate improvements in fault localization accuracies to changes in statistical rankings obtained before and after adding test cases. Having these rules and having a pair of statistical rankings from before and after adding some test cases, we can predict whether test generation should be stopped or continued.

5.2 Approach

In this section, we present our approach to improve statistical debugging for Simulink by generating a *small* number of test cases. Our test generation aims to improve statistical ranking results by maximizing diversity among test cases. An overview of our approach is illustrated by the algorithm in Figure 5.1. As the algorithm shows, our approach uses two subroutines TESTGENERATION and STOPTESTGENERATION to improve the standard fault localization based on statistical debugging (STATISTICALDEBUGGING). Engineers start with an initial test suite *TS* to localize faults in Simulink models (Lines 1-2). Since STATISTICALDEBUGGING requires pass/fail information about individual test cases, engineers are expected to have developed test oracles for *TS*. Our approach then uses subroutine STOPTESTGENERATION to determine whether adding more test cases to *TS* can improve the existing ranking (Line 4). If so, then our approach generates a number of new test cases *newTS* using the TESTGENERATION subroutine (Line 6). The number of generated test cases (i.e., k) is determined by engineers. The new test cases are then passed to the standard statistical debugging to generate a new statistical ranking. Note that this requires engineers to develop test oracle information for the new test cases (i.e., test cases in *newTS*). The iterative process continues until a number of test generation rounds as specified by the input *round* variable are performed, or the STOPTESTGENERATION subroutine decides to stop the test generation process. We present subroutines TESTGENERATION and STOPTESTGENERATION in Sections 5.2.1 and 5.2.2, respectively.

```
SimulinkFaultLocalization()

Input:  - $TS$: An initial test suite
        - $M$ : A Simulink model
        - $round$: The number of test generation rounds
        - $k$: The number of new test cases per round
Output: $rankList$: A statistical debugging ranking

1.  $rankList, TES_{TS} \leftarrow$ StatisticalDebugging($M, TS$)
2.  $initialList \leftarrow rankList$
3.  for $r \leftarrow 0, 1, \ldots, round - 1$ do
4.     if StopTestGeneration($round$, $M$, $initialList$, $rankList$) then
5.        break for-Loop
6.     $newTS \leftarrow$ TestGeneration($TES_{TS}, M, k$)
7.     $TS \leftarrow TS \cup newTS$
8.     $rankList, TES_{TS} \leftarrow$ StatisticalDebugging($M, TS$)
9.  end
10. return $rankList$
```

Figure 5.1. Overview of our Simulink fault localization approach.

5.2.1 Search-based Test Generation

We use *search-based* techniques [Luke, 2015] to generate test cases that improve statistical debugging results. To guide the search algorithm, we define fitness functions that aim to increase diversity of test cases. Our intuition is that diversified test cases are likely to execute varying subsets of Simulink model blocks. As a result, Simulink blocks are likely to take different scores, and hence, the resulting rank groups in the statistical ranking are likely to be smaller. In this section, we first present the fitness functions that are used to guide test generation, and then, we discuss the search-based test generation algorithm. We describe three different alternative fitness functions referred to as *coverage dissimilarity*, *coverage density* and *number of dynamic basic blocks*. Coverage dissimilarity has previously been used for test prioritization [Jiang et al., 2009], and is used in this chapter for the first time to improve fault localization. The two other alternatives, i.e., *coverage density* [Campos et al., 2013] and *number of dynamic basic blocks* [Baudry et al., 2006], have been previously used to improve source code fault localization.

Coverage Dissimilarity. Coverage dissimilarity aims to increase diversity between test execution slices generated by test cases. We use a set-based distance metric known as Jaccard distance [Jaccard, 1901] to define coverage dissimilarity. Given a pair $tes_{tc,o}$ and $tes_{tc',o'}$ of test execution slices, we denote their dissimilarity as $d(tes_{tc,o}, tes_{tc',o'})$ and define it as follows:

$$\mathrm{d}(tes_{tc,o}, tes_{tc',o'}) = 1 - \frac{|tes_{tc,o} \cap tes_{tc',o'}|}{|tes_{tc,o} \cup tes_{tc',o'}|}$$

The coverage dissimilarity fitness function, denoted by fit_{Dis}, is the average of pairwise dissimilarities between every pair of test execution slices in TES_{TS}. Specifically,

$$fit_{Dis}(TS) = \frac{2 \times \sum_{tes_{tc,o}, tes_{tc',o'} \in TES_{TS}} d(tes_{tc,o}, tes_{tc',o'})}{|TES_{TS}| \times (|TES_{TS}| - 1)}$$

The larger the value of $\mathit{fit}_{Dis}(TS)$, the larger the dissimilarity among test execution slices generated by *TS*. For example, the dissimilarity between test execution slices $tes_{t_1,TOut}$ and $tes_{t_2,TOut}$ in Table 3.1 is 0.44. Also, for that example, the average pairwise dissimilarities $\mathit{fit}_{Dis}(TS)$ is 0.71.

Coverage Density. Campos et al [Campos et al., 2013] argue that the accuracy of statistical fault localization relies on the density of test coverage results. They compute the test coverage density as the average percentage of components covered by test cases over the total number of components in the underlying program. We adapt this computation to Simulink, and compute the coverage density of a test suite *TS*, denoted by $p(TS)$, as follows:

$$\mathrm{p}(TS) = \frac{1}{|TES_{TS}|} \sum_{tes_{tc,o} \in TES_{TS}} \frac{|tes_{tc,o}|}{|static\_slice(o)|}$$

That is, our adaptation of coverage density to Simulink computes, for every output o, the average size of test execution slices related to o over the static backward slice of o. Note that a test execution slice related to output o is always a subset of the static backward slice of o. Low values of $p(TS)$ (i.e., close to zero) indicate that test cases cover small parts of the underlying model, and high values (i.e., close to one) indicate that test cases tend to cover most parts of the model. According to Campos et al [Campos et al., 2013], a test suite whose coverage density is equal to 0.5 (i.e., neither low nor high) is more capable of generating accurate statistical ranking results. Similar to Campos et al [Campos et al., 2013], we define the coverage density fitness function as $fit_{Dens}(TS) = |0.5 - p(TS)|$ and aim to minimize $\mathit{fit}_{Dens}(TS)$ to obtain more accurate ranking results.

Number of Dynamic Basic Blocks. Given a test suite *TS* for fault localization, a *Dynamic Basic Block (DBB)* [Baudry et al., 2006] is a subset of program statements such that for every test case $tc \in TS$, all the statements in *DBB* are either all executed together by *tc* or none of them is executed by *tc*. According to [Baudry et al., 2006], a test suite that can partition the set of statements of the program under analysis into a large number of dynamic basic blocks is likely to be more effective for statistical debugging. In our work, we (re)define the notion of DBB for Simulink models based on test execution slices. Formally, a set *DBB* is a dynamic basic block iff $DBB \subseteq Nodes$ and for every test execution slice $tes \in TES_{TS}$, we have either $DBB \subseteq tes$ or $DBB \cap tes = \emptyset$. For a given set TES_{TS} of test execution slices obtained by test suite *TS*, we can partition the set *Nodes* of Simulink model blocks into a number of *disjoint* dynamic basic blocks $DBB_1, \ldots, DBB_l$. Our third fitness function, which is defined based on dynamic basic blocks and is denoted by $fit_{dbb}(TS)$, is defined as the number of dynamic basic blocks produced by a given test suite *TS*, i.e., $fit_{dbb}(TS) = l$. The larger the number dynamic basic blocks, the better the quality of a test suite *TS* for statistical debugging.

Test generation algorithm. Having defined the fitness functions, we now define our search-based test generation algorithm (i.e. TESTGENERATION in Figure 5.1). The TESTGENERATION algorithm is shown in Figure 5.2 and generates new test cases based on any of our three fitness functions. The algorithm adapts a single-state search optimizer [Luke, 2015]. In particular, it builds on the Hill-Climbing with Random Restarts (HCRR) algorithm [Luke, 2015]. We chose to build on HCRR because, in our previous work on testing Simulink models [Matinnejad et al., 2015a], HCRR was able to produce the best optimized test cases among other single-state optimization algorithms. Computation of all the three fitnesses we described earlier rely on test execution slices. To obtain test execution slices, we need to execute test cases on Simulink models. This makes our fitness computation expen-

```
Algorithm. TESTGENERATION
Input:  - $TES_{TS}$: The set of test execution slices
        - $M$ : The Simulink model
        - $k$: The number of new test cases
Output: $newTS$: A set of new test cases

1.  $TS_{curr} \leftarrow$ Generate $k$ test cases $tc_1, \ldots, tc_k$ (randomly)
2.  $TES_{curr} \leftarrow$ Generate the union of the test execution slices of
              the $k$ test cases in $TS_{curr}$
3.  $fit_{curr} \leftarrow$ ComputeFitness $(TES_{curr} \cup TES_{TS}, M)$
4.  $fit_{best} \leftarrow fit_{curr}$;  $TS_{best} \leftarrow TS_{curr}$
5.  repeat
6.    while ($time$ != $restartTime$ )
7.      $TS_{new} \leftarrow$ Mutate the $k$ test cases in $TS_{curr}$
8.      $TES_{new} \leftarrow$ Generate the union of the test execution slices of
                the $k$ test cases in $TS_{new}$
9.      $fit_{new} \leftarrow$ ComputeFitness $(TES_{new} \cup TES_{TS}, M)$
10.     if ($fit_{new}$ is better than $fit_{curr}$ )
11.         $fit_{curr} \leftarrow fit_{new}$;  $TS_{curr} \leftarrow TS_{new}$
12.   end
13.   if ($fit_{curr}$ is better than $fit_{best}$ )
14.     $fit_{best} \leftarrow fit_{curr}$;  $TS_{best} \leftarrow TS_{curr}$
15.   $TS_{curr} \leftarrow$ Generate $k$ test cases $tc_1, \ldots, tc_k$ (randomly)
16. until the time budget is reached
17. return $TS_{best}$
```

Figure 5.2. Test case generation algorithm.

sive. Hence, in this chapter, we rely on single-state search optimizers as opposed to population-based search techniques.

The algorithm in Figure 5.2 receives as input the existing set of test execution slices TES_{TS}, a Simulink model M, and the number of new test cases that need to be generated (k). The output is a test suite (*newTS*) of k new test cases. The algorithm starts by generating an initial randomly generated set of k test cases TS_{curr} (Line 1). Then, it computes the fitness of TS_{curr} (Line 3) and sets TS_{curr} as the current best solution (Line 4). The algorithm then searches for a best solution through two nested loops: (1) *The internal loop* (Lines 6 to 12). This loop tries to find an optimized solution by locally tweaking the existing solution. That is, the search in the inner loop is *exploitative*. The mutation operator in the inner loop generates a new test suite by tweaking the individual test cases in the current test suite and is similar to the tweak operator used in our earlier work [Matinnejad et al., 2015b]. (2) *The external loop* (Lines 5 to 16). This loop tries to find an optimized solution through random search. That is, the search in the outer loop is *explorative*. More precisely, the algorithm combines an exploitative search with an explorative search. After performing an exploitative search for a given amount of time (i.e., *restartTime*), it restarts the search and moves to a randomly selected point (Line 15) and resumes the exploitative search from the new randomly selected point. The algorithm stops after it reaches a given time budget (Line 15).

We discuss two important points about our test generation algorithm: (1) Each candidate solution in our search algorithm is a test suite of size k. This is similar to the approach taken in the *whole test suite generation* algorithm proposed by Fraser and Arcuri in [Fraser and Arcuri, 2013]. The reason we use a whole test suite generation algorithm instead of generating test cases individually is that computing fitnesses for one test case and for several test cases takes almost the same amount of time. This is because, in our work, the most time consuming operation is to load a Simulink model. Once the model is loaded, the time required to run several test cases versus one test case is not very different. Hence, we decided to generate and mutate the k test cases at the same time. (2) Our algorithm does not require test oracles to generate new test cases. Note that computing fit_{Dis} and fit_{dbb} only requires test execution slices without any pass/fail information. To compute fit_{Dens}, in addition to test execution slices, we need static backward slices that can be obtained from Simulink models. Test oracle information for the k new test cases is only needed after test generation in subroutine STATISTICALDEBUGGING (see Figure 5.1) when a new statistical ranking is computed. In the next section, we discuss the STOPTESTGENERATION subroutine (see Figure 5.1) that allows us to stop test generation before performing all the test generation rounds when we can predict situations where test generation is unlikely to improve the fault localization.

5.2.2 Stopping Test Generation

As noted in the literature [Campos et al., 2013], adding test cases does not always improve statistical debugging results. Given that in our context test oracles are expensive, we provide a strategy to stop test generation when adding new test cases is unlikely to bring about noticeable improvements in the fault localization results. Our STOPTESTGENERATION subroutine is shown in Figure 5.3. It has two main parts: In the first part (Lines 1–6), it tries to determine if the decision about stopping test generation can be made only based on the characteristics of *newList* (i.e., the latest generated ranked list) and static analysis of Simulink models. For this purpose, it computes Simulink super blocks and compares the top ranked groups of *newList* with Simulink super blocks. In the second part (Lines 7-10), our algorithm relies on a predictor model to make a decision about further rounds of test generation. We build the predictor model using supervised learning techniques (i.e., *decision trees* [Olshen et al., 1984]) based on the following three features: (1) the current test generation round, (2) the *SetDistance* between the latest ranked list and the initial ranked list, and (3) the *OrderingDistance* between the latest ranked list and the initial ranked list. Below, we first introduce Simulink super blocks. We will then introduce *SetDistance* and the *OrderingDistance* that are used as input features for our predictor model. After that, we describe how we build and use our decision tree predictor model.

Super blocks. Given a Simulink model $M = (Nodes, Links, Inputs, Outputs)$, we define a *super block* as the largest set $B \subseteq Nodes$ of (atomic) Simulink blocks such that for every test case tc and every output $o \in Outputs$, we have either $B \subseteq tes_{tc,o}$ or $B \cap tes_{tc,o} = \emptyset$. The definition of super block is very similar to the definition of dynamic basic blocks (DBB) discussed in Section 5.2.1. The only difference is that dynamic basic blocks are defined with respect to the test execution slices generated by a given test suite, while super blocks are defined with respect to test execution slices that can be generated by any potential test case. Hence, dynamic basic blocks can be computed *dynamically* based on test execution slices obtained by the current test suite, whereas super blocks are computed by *static analysis* of the structure of Simulink models. In order to compute super blocks, we identify conditional (control) blocks in the given Simulink model. Each conditional block has an incoming control link and a number of incoming data links. Corresponding to each conditional block, we create some branches by matching each incoming data link with the conditional branch link. We then remove the conditional block and replace it with the new branches. This allows us to obtain a behaviorally equivalent Simulink model with no conditional blocks. We further remove parallel branches by replacing them with their equivalent sequential linearizations. We then use the resulting

```
STOPTESTGENERATION()

Input:  - $r$: The index of the latest test generation round
        - $M$: The underlying Simulink model
        - initialList: A ranked list obtained using an initial test suite
        - newList: A ranked list obtained at round $r$ after some
            test cases are added to the initial test suite
Output: result: Test generation should be stopped if result is true

1.  Let $rg_1, \ldots, rg_N$ be the top $N$ rank groups in newList
2.  Identify Simuilnk superblocks $B_1, \ldots, B_m$ in the set $rg_1 \cup \ldots \cup rg_N$
3.  if for every $rg_i$ $(1 \leq i \leq N)$ there is a $B_j$ $(1 \leq j \leq m)$ s.t. $rg_i = B_j$ then
4.      return true
5.  if $r = 0$ then
6.      return false
7.  $m_1$ = ComputeSetDistance(initialList, newList)
8.  $m_2$ = ComputeOrderingDistance(initialList, newList)
9.  result = Prediction($m_1$, $m_2$, $r$)
10. return result
```

Figure 5.3. The STOPTESTGENERATION subroutine used in our approach (see Figure 5.1).

Simulink model to partition the set *Nodes* into a number of *disjoint* super blocks $B_1, \ldots, B_l$.

We briefly discuss the important characteristics of super blocks. Let *rankList* be a ranked list obtained based on statistical debugging, and let *rg* be a ranked group in *rankList*. Note that *rg* is a set as the elements inside a ranked group are not ordered. For any super block B, if $B \cap rg \neq \emptyset$ then $B \subseteq rg$. That is, the blocks inside a super block always appear in the same ranked group, and cannot be divided into two or more ranked groups. Furthermore, if $rg = B$, we can conclude that the ranked group *rg* cannot be decomposed into smaller ranked groups by adding more test cases to the test suite used for statistical debugging.

Features for building our predictor model. We describe the three features used in our predictor models. The first feature is the test generation round. As shown in Figure 5.1, we generate test cases in a number of consecutive rounds. Intuitively, adding test cases at the earlier rounds is likely to improve statistical debugging more compared to the later rounds. Our second and third features (i.e., *SetDistance* and *OrderingDistance*) are similarity metrics comparing the latest generated rankings (at the current round) and the initial rankings. These two metrics are formally defined below.

Let *initialList* be the ranking generated using an initial test suite, and let *newList* be the latest generated ranking. Let $rg_1^{new}, \ldots, rg_m^{new}$ be the ranked groups in *newList*, and $rg_1^{initial}, \ldots, rg_{m'}^{initial}$ be the ranked groups in *initialList*. Our *SetDistance* feature computes the dissimilarity between the top-N ranked groups of *initialList* and *newList* using the *intersection metric* [Fagin et al., 2003]. We focus on comparing the top N ranked groups because, in practice, the top ranked groups are primarily inspected by engineers. We compute the *SetDistance* based on the average of the overlap between the top-N ranked groups of the two ranked lists. Formally, we define the *SetDistance* between *initialList* and *newList* as follows.

$$IM(initialList, newList) = \frac{1}{N} \sum_{k=1}^{N} \frac{|\{\bigcup_{i=1}^{k} rg_i^{initial}\} \cap \{\bigcup_{i=1}^{k} rg_i^{new}\}|}{|\{\bigcup_{i=1}^{k} rg_i^{initial}\} \cup \{\bigcup_{i=1}^{k} rg_i^{new}\}|}$$
$$SetDistance(initialList, newList) = 1 - IM(initialList, newList)$$

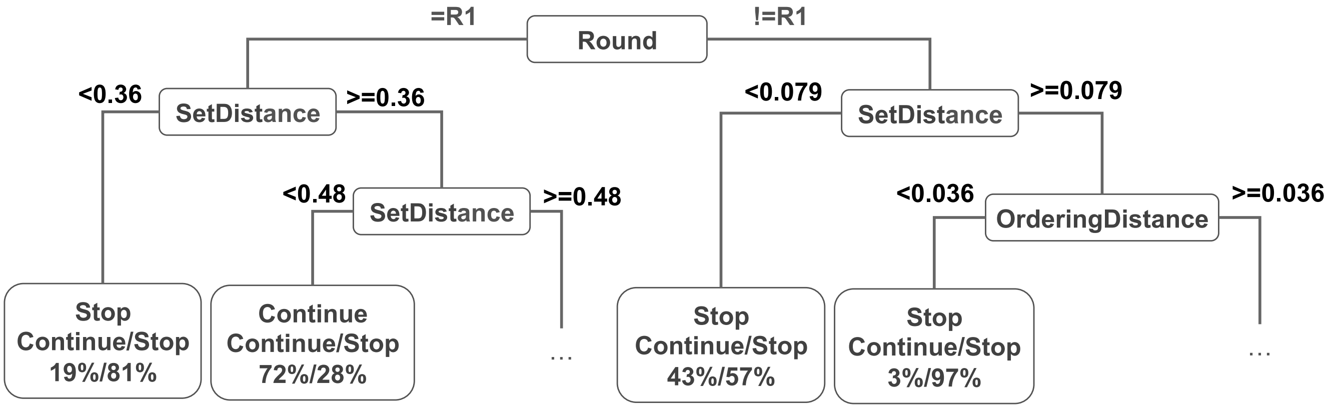


Figure 5.4. A snapshot example of a decision tree.

The larger the *SetDistance*, the more differences exist between the top-*N* ranked groups of *initialList* and *newList*.

Our third feature is *OrderingDistance*. Similar to *SetDistance*, the *OrderingDistance* feature also attempts to compute the dissimilarity between the top-*N* ranked groups of *initialList* and *newList*. However, in contrast to *SetDistance*, *OrderingDistance* focuses on identifying changes in pairwise orderings of blocks in the rankings. In particular, we define *OrderingDistance* based on *Kendall Tau Distance* [Kendall, 1938] that is a well-known measure for such comparisons. This measure computes the dissimilarity between two rankings by counting *the number of discordant pairs* between the rankings. A pair b and b' is discordant if b is ranked higher than b' in *newList* (respectively, in *initialList*), but not in *initialList* (respectively, in *newList*). In our work, in order to define the *OrderingDistance* metric, we first create two sets *initialL* and *newL* based on *initialList* and *newList*, respectively: *initialL* is the same as *initialList* except that all the blocks that do not appear in the top-*N* ranked groups of neither *initialList* nor *newList* are removed. Similarly, *newL* is the same as *newList* except that all the blocks that do not appear in the top-*N* ranked groups of neither *newList* nor *initialList* are removed. Note that *newL* and *initialL* have the same number blocks. We then define the *OrderingDistance* metric as follows:

$$OrderingDistance(newL, initialL) = \frac{\text{\# of Discordant Pairs}}{(|newL| \times (|newL|-1))/2}$$

The larger the *OrderingDistance*, the more differences exist between the top-*N* ranked groups of *initialList* and *newList*.

Prediction model. Our prediction model builds on an intuition that by comparing statistical rankings obtained at the current and previous rounds of test generation, we may be able to predict whether further rounds of test generation are useful or not. We build a prediction model based on the three features discussed above (i.e., the current round, *SetDistance*, *OrderingDistance*). We use supervised learning methods, and in particular, decision trees [Olshen et al., 1984]. The prediction model returns a binary answer indicating whether the test generation should stop or not. To build the prediction model, we use historical data consisting of statistical rankings obtained during a number of test generation rounds and *fault localization accuracy* results corresponding to the statistical rankings. When such historical data is not available the prediction model always recommends that test generation should be continued. After applying our approach (Figure 5.1) for a number of rounds, we gradually obtain the data that allows us to build a more effective prediction model that can recommend to stop test generation as well. Specifically, suppose *rankList* is a ranking obtained at round *r* of our approach (Figure 5.1), and suppose *initList* is a ranking obtained initially before generating test cases (Figure 5.1). The accuracy of fault localization for *rankList* is *the maximum number of blocks inspected* to find a fault when engineers use *rankList* for inspection. To build our decision tree, for each *rankList* computed by our approach in Figure 5.1, we obtain the triple $I = (r, SetDistance(initList, rankList), OrderingDistance(initList, rankList))$. We then compute the maximum fault localization accuracy improvement that we can achieve if we proceed with test generation from round *r* (the current round) until the last round of our algorithm in Figure 5.1. We denote the maximum fault localization accuracy improvement by $Max\_ACC_r(rankList)$. We then label the triple *I* with `Continue`, indicating that test generation should continue, if $Max\_ACC_r(rankList)$ is more than a threshold (*THR*); and with `Stop`, indicating that test generation should stop, if $Max\_ACC_r(rankList)$ is less than the threshold (*THR*). Note that *THR* indicates the minimum accuracy improvements that engineers expect to obtain to be willing to undergo the overhead of generating new test cases.

Having obtained triples *I* labelled with `Stop` or `Continue`, we build our decision tree model (prediction model). Decision trees are composed of leaf nodes, which represent *partitions*, and non-leaf nodes, which represent *decision variables*. A decision tree model is built by partitioning the set of input triples in a stepwise manner aiming to create partitions with increasingly more homogeneous labels (i.e., partitions in which the majority of triples are labelled either by `Stop` or by `Continue`). The larger the difference between the number of triples with `Stop` and `Continue` in a partition, the more homogeneous that partition is. Decision variables (i.e., non-leaf node) in our decision tree model represent logical conditions on the input features (i.e., *r*, *SetDistance*, or *OrderingDistance*). Figure 5.4 shows a fragment of our decision tree model. For example, the model shows, among the triples satisfying r = `R1` and $SetDistance < 0.36$ conditions, 81% are labelled with `Stop` and 19% are labelled with `Continue`.

We stop splitting partitions in our decision tree model if the number of triples in the partitions is smaller than α, or the percentage of the number of triples in the partitions with the same label is higher than β. In this work, we set α to 50 and β to 95%, i.e., we do not split a partition whose size is less than 50, or at least 95% of its elements have the same label.

Stop Test Generation Algorithm. The STOPTESTGENERATION() algorithm starts by identifying the super blocks in *newList*, the latest generated ranking (Line 2). If it happens that the top-*N* ranked groups in *newList* all comprise a single super block, then test generation stops (Line 3-4), because such ranking cannot be further refined by test generation. If we are in the first round (i.e., $r = 0$), the algorithm returns *false*, meaning that test generation should continue. For all other rounds, we use the decision tree prediction model. Specifically, we compute the *SetDistance* and *OrderingDistance* features corresponding to *newList*, and pass these two values as well as *r* (i.e., the round) to the prediction model. The prediction model returns *true*, indicating that test generation should be stopped, if the three input features satisfy a sequence of conditions leading to a (leaf) partition where at least 95% of the elements in that partition are labelled `Stop`. Otherwise, our prediction model returns *false*, indicating that test generation should be continued. For example, assuming the decision tree in Figure 5.4 is our prediction model, we stop test generation only if we are not in round one, *SetDistance* is greater than or equal to 0.079, and *OrderingDistance* is less than 0.036. This is because, in Figure 5.4, these conditions lead to the leaf partition with 97% stop-labelled elements.

5.3 Empirical Evaluation

5.3.1 Research Questions

RQ1. [Evaluating and comparing different test generation fitness heuristics] *How is the fault localization accuracy impacted when we apply our search-based test generation algorithm in Figure 5.2 with our three selected fitness functions (i.e., coverage dissimilarity (f_{Dis}), coverage density (f_{Dens}), and number of dynamic basic blocks (f_{dbb}))?* We report the fault localization accuracy of a ranking generated by an initial test suite compared to that of a ranking generated by a test suite extended using our algorithm in Figure 5.2 with a small number of test cases. We further compare the fault localization accuracy improvement when we use our three alternative fitness functions, and when we use a random test generation strategy not guided by any of these fitness functions.

RQ2. [Evaluating impact of adding test cases] *How does the fault localization accuracy change when we apply our search-based test generation algorithm in Figure 5.2?* We note that adding test cases does not always improve the fault localization accuracy [Campos et al., 2013]. With this question, we investigate how often fault localization accuracy improves after adding test cases. In particular, we apply our approach in Figure 5.1 without calling the STOPTESTGENERATION subroutine, and identify how often subsequent rounds of test generation do not lead to fault localization accuracy improvement.

RQ3. [Effectiveness of our STOPTESTGENERATION subroutine] *Does our* STOPTESTGENERATION *subroutine help stop generating additional test cases when they do not improve the fault localization accuracy?* We investigate whether the predictor model used in the STOPTESTGENERATION subroutine can stop test generation when adding test cases is unlikely to improve the fault localization accuracy, or when the improvement that the test cases bring about is small compared to the effort required to develop their test oracles.

5.3.2 Experiment Settings

In this section, we describe the industrial subjects, test suites and test oracles used in our experiments.

Industrial Subjects. In our experiment, we use three Simulink models referred to as *MA*, *MZ* and *MGL*, and developed by Delphi Automotive [Delphi Automotive LLP, 2017]. Table 5.1 shows the number of subsystems, atomic blocks, links, and inputs and outputs of each model. Note that the models that we chose are representative in terms of size and complexity among the Simulink models developed at Delphi. Further, these models include about ten times more blocks than the publicly available Simulink models from the Mathworks model repository [MathWorks, b].

Table 5.1. Key information about industrial subjects.

Model Name	#Subsystem	#Blocks	#Links	#Inputs	#Outputs	#Faulty version
MA	37	680	663	12	8	20
MZ	65	833	806	13	7	20
MGL	33	742	730	19	9	20

We asked a Delphi engineer to seed 20 realistic and typical faults into each model. We have provided detailed descriptions of the seeded faults in [Liu, 2017]. In total, we generated 60 faulty versions (one fault per each faulty version). We ensured that the faults were of different types and were seeded into different parts of the models. All experiment data and scripts are available in [Liu, 2017].

Test Suite and Test Oracles. We generated three initial test suites (i.e., *TS* in Figure 5.1) for MA, MZ and MGL using Adaptive Random Testing [Chen et al., 2005]. Adaptive random testing is a black box and lightweight test generation strategy that distributes test cases evenly within valid input ranges, and thus, helps ensure diversity among test cases. Given that in our work we assume test oracles are manual, we aim to generate test suites that are not large. However, the test suites should be large enough to generate a meaningful statistical ranking. Hence, at least some test cases in the test suite exhibit failures. In our work, we chose to use initial test suites with size 10. To enable the full automation of our experiments, we used the fault-free versions of our industrial subjects as test oracles. On average, our initial test suites covered 75.5% of the structure of the faulty models.

Experiment Design. To answer our research questions, we applied our approach to the faulty versions of our three models, in total 60 faulty versions. We refer to the test generation algorithm in Figure 5.2 as HCRR since it builds on the HCRR search algorithm. We refer to HCRR when it is used with fitness functions f_{Dis}, f_{Dens} and f_{dbb} as HCRR-Dissimilarity, HCRR-Density and HCRR-DBB, respectively. We set both the number of new test cases per round (i.e., *k* in Figure 5.1), and the number of rounds (i.e., *round* in Figure 5.1) to five. That is, in total, we generate 25 new test cases by applying our approach. We applied our three alternative HCRR algorithms to our 60 faulty versions. We ran each HCRR algorithm for 45 minutes with two restarts. To account for randomness of the search algorithms, we repeat our experiments for ten times (i.e., ten trials). Further, to compute input features for our stopping criteria setting, we set *N* (in Figure 5.3) to five. We ran our experiment on a high performance computing platform [Varrette et al., 2014] with 2 clusters, 280 nodes, and 3904 cores. Our experiment were executed on different nodes of a cluster with Intel Xeon L5640@2.26GHz processor. In total, our experiment (using a single node 4 cores) required 6750 hours. Most of the experiment time was used to execute the generated test cases in Simulink. In total, we generated and executed 129000, 159000, and 120000 test cases for *MA MZ*, and *MGL*, respectively.

5.3.3 Evaluation Metrics

We evaluate the accuracy of the rankings generated at different rounds of our approach using the following metrics [Cleve and Zeller, 2005, Jones and Harrold, 2005, Liu et al., 2005, Lucia et al., 2014, Parnin and Orso, 2011, Renieris and Reiss, 2003]: the *absolute number of blocks inspected* to find faults, and the *proportion of faults localized* when engineers inspect fixed numbers of the top most suspicious blocks. The former was already discussed for prediction models in Section 5.2.2. The proportion of faults localized is the proportion of localized faults over the total number of faults when engineers inspect a fixed number of the top most suspicious blocks from a ranking.

5.3.4 Experiment Results

RQ1. [Evaluating and comparing different test generation fitness heuristics] Figure 5.5 compares the fault localization results after applying HCRR-DBB, HCRR-Density and HCRR-Dissimilarity algorithms to generate 25 test cases (five test cases in five rounds) with the fault localization results obtained before applying these algorithms (i.e., Initial) and with the fault localization results obtained after generating 25 test cases randomly (i.e., Random). In particular, in Figure 5.5(a), we compare the distributions of the maximum number of blocks inspected to locate faults (i.e. accuracy) in our 60 faulty versions when statistical rankings are generated based on the initial test suite (i.e. Initial), or after using HCRR-DBB, HCRR-Density, HCRR-Dissimilarity and Random test generation to add 25 test cases to the initial test suite. Each point in Figure 5.5(a) represents fault localization accuracy for one run of one faulty version. According to Figure 5.5(a), before applying our approach (i.e., Initial), engineers on average need to inspect at most 76 blocks to locate faults. When in addition to the initial test suite, we use 25 *randomly* generated test cases, the maximum number of blocks inspected decreases to, on average, 62 blocks. Finally, engineers need to inspect, on average, 42.4, 44 and 42.8 blocks if they use the rankings generated by HCRR-DBB, HCRR-Density and HCRR-Dissimilarity, respectively. We performed non-parametric pairwise Wilcoxon signed-rank tests to check whether the improvement on the number of blocks inspected is statistically significant. The results show that the fault localization accuracy distributions obtained by HCRR-DBB, HCRR-Density and HCRR-Dissimilarity are significantly lower (better) than those obtained by Random and Initial (with *p*-value$<$0.0001).

Similarly, Figure 5.5(b) shows the proportion of faults localized when engineers inspect a fixed number of blocks in the rankings generated by Initial, and after generating 25 test cases with HCRR-DBB, HCRR-Density, HCRR-Dissimilarity, and Random. Specifically, the X-axis shows the number of top ranked blocks (ranging from 10 to 80), and the Y-axis shows the proportion of faults among a fixed number of top ranked blocks in the generated rankings. Note that, in Figure 5.5(b), the maximum number of blocks inspected (X-axis) is computed as an average over ten trials for each faulty version. According to Figure 5.5(b), engineers can locate faults in 13 out of 60 (21.67%) faulty versions when they inspect at most 10 blocks in the rankings generated by any of our techniques i.e., HCRR-DBB, HCRR-Density and HCRR-Dissimilarity. However, when test cases are generated randomly, by inspecting the top 10 blocks, engineers can locate faults in only 3 out of 60 (5%) faulty versions. As for the rankings generated by the initial test suite, no faults can be localized by inspecting the top 10 blocks. Using HCRR-DBB, HCRR-Density and HCRR-Dissimilarity, on average, engineers can locate 50% of the faults in the top 25 blocks of each ranking. In contrast, when engineers use the initial test suite or a random test generation strategy, in order to find 50% of the faults, they need to inspect, on average, 50 blocks in each ranking.

In summary, the test cases generated by our approach are able to help significantly improve the accuracy of fault localization results. In particular, by adding a small number of test cases (i.e., only 25 test cases), we are able to reduce the average number of blocks that engineers need to inspect to find a fault from 76 to 43 blocks (i.e., 43.4% reduction). Further, we have shown that the fault localization accuracy results obtained based on HCRR-DBB, HCRR-Density and HCRR-Dissimilarity are significantly better than those obtained by a random test generation strategy. Specifically, with Random test generation, engineers need to inspect an average of 62 blocks versus an average of 43 blocks when HCRR-DBB, HCRR-Density and HCRR-Dissimilarity are used.

RQ2. [Evaluating impact of adding test cases] We evaluate the fault localization accuracy of the ranking results obtained at each test generation round. In particular, we computed the fault localization accuracy of rankings obtained by applying HCRR-DBB, HCRR-Density and HCRR-

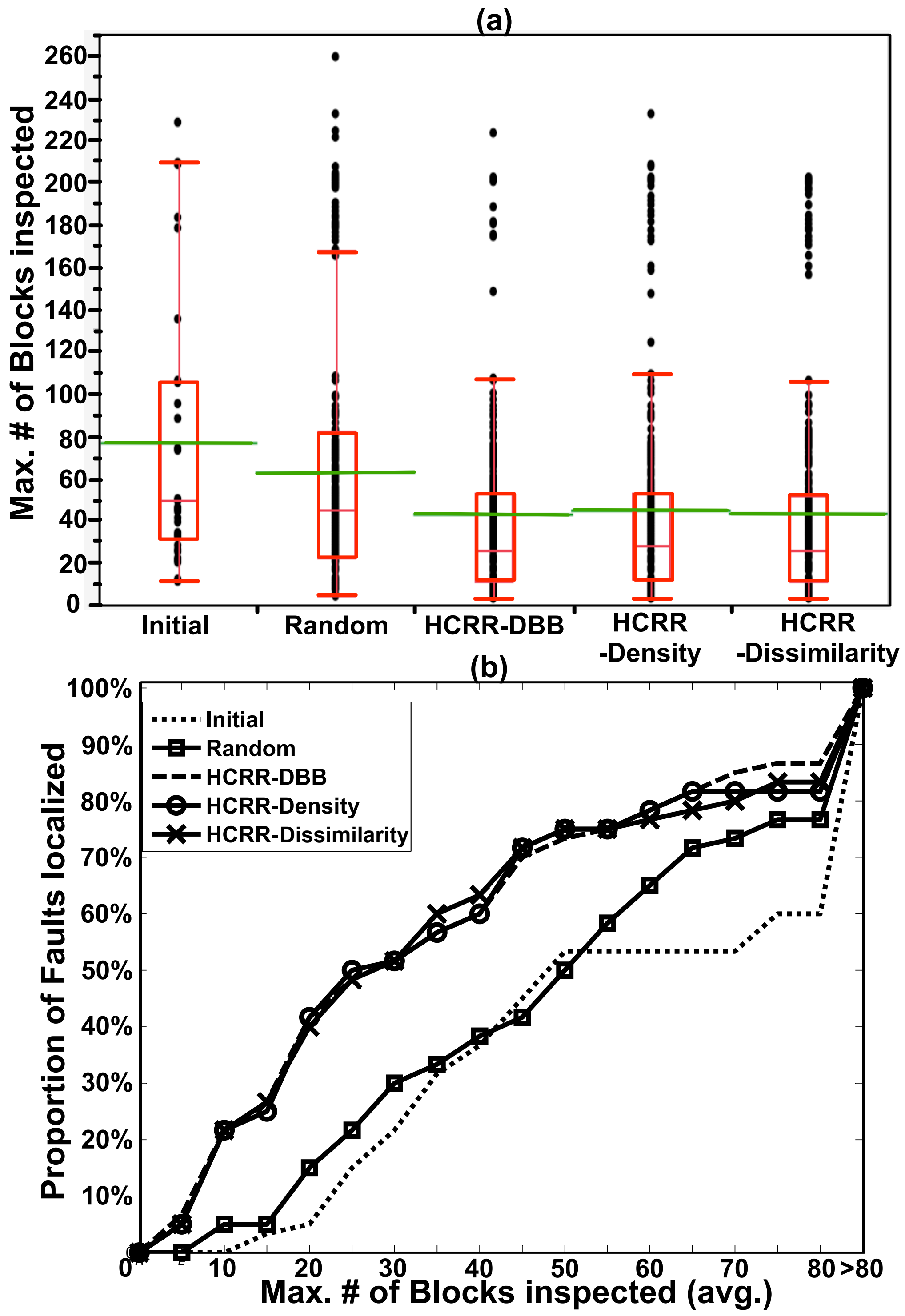


Figure 5.5. Comparing the number of blocks inspected (a) and the proportion of faults localized (b) before and after applying HCRR-DBB, HCRR-Dissimilarity and HCRR-Density, and with Random test generation (i.e., Random).

Dissimilarity to our 60 faulty versions from round one to five where at each round five new test cases are generated. Recall that we have repeated 10 times each application of our technique to each faulty model. That is, in total, we have 1800 trials (60 faulty versions $\times$ 3 algorithms $\times$ 10 runs). Among these 1800 trials, we observed that, as we go from round one to round five, in 953 cases (i.e., 53%), the fault localization accuracy improves at every round; in 803 cases (i.e., 44.6%), the accuracy improves at some (but not all) rounds; and in 44 cases (i.e., 2.4%), the accuracy never improves at any of the rounds from one to five.

To explain why adding new test cases does not always improve fault localization accuracy, we investigate the notion of Coincidentally Correct Test cases (CCT) for Simulink (discussed in Chapter 3). CCTs are test execution slices that execute faulty blocks but do not result in failure. We note that as we add new test cases, the number of CCTs may either stay the same or increase. In the former case, the fault localization accuracy either stays the same or improves. However, in the latter case, the accuracy changes will be unpredictable.

In summary, adding test cases may not always improve fault localization accuracy. Hence, it is important to have mechanisms to help engineers stop test generation when it is unlikely to be beneficial for fault localization.

RQ3. [Effectiveness of our STOPTESTGENERATION subroutine] In order to generate the prediction model used in the STOPTESTGENERATION subroutine, we consider all the statistical ranking results obtained by applying the five rounds of test generation to the 60 faulty versions as well as the corresponding accuracy results. We randomly divide the results into three sets, and use one of these sets to build the decision tree prediction model (i.e., as a training set). The other two sets are used to evaluate the decision tree prediction model (i.e., as test sets). Following a standard cross-validation procedure, we follow this process three times so that each set is used as the training set at least once. To build these models, we set *THR* = 15 (i.e., the threshold used to determine the `Stop` and the `Continue` labels in Section 5.2.2). That is, engineers are willing to undergo the overhead of adding new test cases if the fault localization accuracy is likely to improve by at least 15 blocks. Figure 5.6(a) shows the fault localization accuracy results (i.e., the maximum number of blocks inspected) obtained by our three test generation algorithms (HCRR-DBB, HCRR-Density, and HCRR-Dissimilarity) and when the STOPTESTGENERATION subroutine is used with the three decision tree prediction models generated by cross-validation. These results are shown in columns with `With stop` label. Figure 5.6(a), further, shows the accuracy results obtained by applying the five rounds *without* using STOPTESTGENERATION in columns labelled `Without stop`. In addition, Figure 5.6(b) shows the number of new test cases generated by HCRR-DBB, HCRR-Density and HCRR-Dissimilarity when we applied the STOPTESTGENERATION subroutine. Note that we generate 25 test cases in five rounds without STOPTESTGENERATION.

According to Figure 5.6, we are able to obtain almost the same fault localization accuracy with considerably fewer number of new test cases when we use the STOPTESTGENERATION subroutine compared to when we do not use it. In particular, on average, when we use the STOPTESTGENERATION subroutine, the fault localization accuracies obtained for HCRR-DBB, HCRR-Dissimilarity and HCRR-Density are 47.3, 47.9 and 50.4, respectively. In contrast, without the STOPTESTGENERATION subroutine, the fault localization accuracies obtained for HCRR-DBB, HCRR-Dissimilarity and HCRR-Density are 43, 43.4 and 45.1, respectively. We note that these accuracies are obtained by only

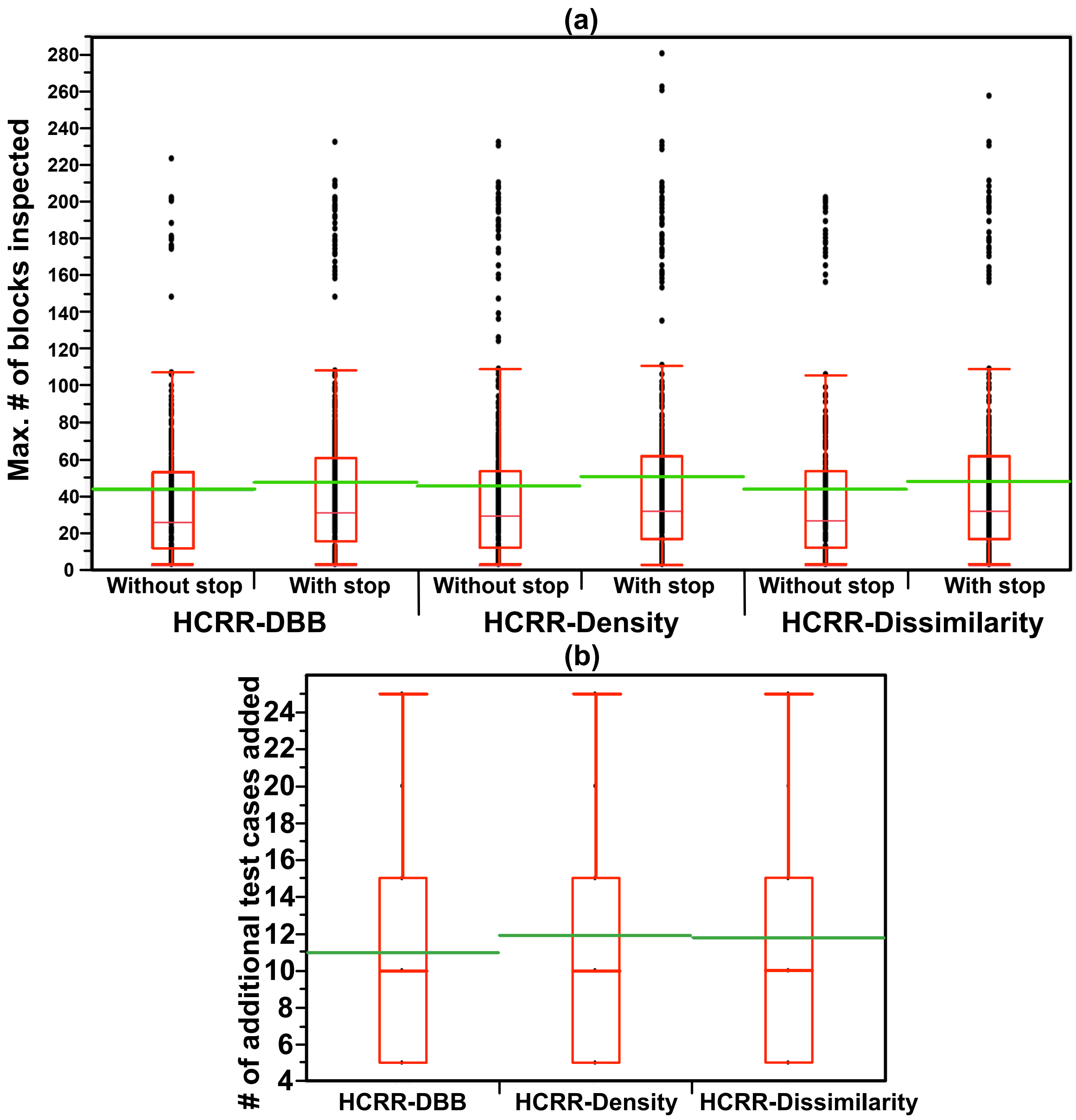


Figure 5.6. The maximum number of blocks inspected and the number of new test cases added when we applied STOPTESTGENERATION on the rankings generated using HCRR-DBB, HCRR-Density, and HCRR-Dissimilarity based on the predictor models obtained for three different validation sets.

generating, on average, 11 test cases for HCRR-DBB, and 12 test cases for both HCRR-Density and HCRR-Dissimilarity. We have also repeated our experiments for $THR = 10$. The results for $THR = 10$ show that the average fault localization accuracies improve by one to two blocks while the number of new test cases also increases by one or two when compared with the results for $THR = 15$.

In summary, our approach identifies situations where adding new test cases does not improve fault localization results. When engineers use the STOPTESTGENERATION subroutine, they need to inspect a few more blocks (i.e., around five blocks), on average, but the number of test cases, and hence the test oracle cost, reduces by more than half (i.e., 52% to 56% fewer test cases).

5.4 Conclusion

In this chapter, we improve fault localization accuracy for Simulink models by extending an existing test suite with a small number of test cases. The latter requirements is very important in contexts where running and analyzing test case is expensive, such as with embedded systems. Our approach has two components: (1) A search-based test generation algorithm that aims to increase test suite diversity, and (2) a predictor model that predicts if additional test cases are likely to help improve fault localization accuracy. Our work is driven by an important consideration that in some situations, test oracles are manual and hence expensive, or running test cases is expensive. As a result, we assess our test generation technique for small test suite sizes, and use our predictor models to avoid generating additional test cases when they cannot lead to substantial improvement justifying their incurred overhead. Our results show that our test generation technique significantly improves the accuracy of fault localization for small test suite sizes, and further, our prediction model is able to maintain a similar fault localization accuracy while reducing the average number of newly generated test cases by more than half.

In future, we intend to study fault localization for evolving Simulink models. A recent study of industrial Simulink models indicates a strong co-evolution relation between changes in models and in their corresponding test suites [Rapos and Cordy, 2016]. We plan to investigate how such relations can be used to generate test suites that lead to effective Simulink fault localization, especially, when models are subject to frequent changes.

Chapter 6

A Fault Localization Tool for Simulink Models

To better support debugging Simulink models, we develop a tool called Simulink Fault Localization tool (*SimFL*) which implements the techniques discussed in Chapter 3, 4, and 5. In this chapter, we present *SimFL*.

Organization. This chapter is organized as follows. We first provide the relevant background and an overview of different features supported by *SimFL* in Section 6.1. Implementations are discussed in Section 6.2.

6.1 Tool Overview & Features

Figure 6.1 shows an overview of *SimFL*. The process of *SimFL* consists of four steps. The first step of *SimFL* is *Test Case Execution*. In this step, *SimFL* calls Matlab/Simulink to execute the input test suite based on a Matlab executable script we devised in *SimFL*. We then collect test coverage information and save it into a .csv file for later steps. The second step, *Super Block Computation*, is optional (shown by dashed line in Figure 6.1). Given a list of model outputs, *SimFL* computes the super block information based on the static analysis of Simulink models (see Section 5.2.2). The third step in *SimFL* is *Fault Localization*. In this step, *SimFL* takes the test coverage information as well as the test oracle data (the pass/fail data for test cases) as inputs, computes the test execution slices and generates ranked lists based on a selected statistical formula. The fourth step (i.e., *Test Suite Extension*) is necessary when the users are not satisfied with the currently generated ranking. In this step, *SimFL* invokes a search-based technique to generate a number of new test cases to improve this ranked list. These steps can be applied independently or sequentially. Moreover, *SimFL* can be used once or iteratively and the whole process can stop when the debugging budget runs out or the ranking result is unlikely to be improved by adding new test cases. In the rest of this section, we elaborate each step of the process shown in Figure 6.1.

6.1.1 Test Suite Execution

"*Test Suite Execution*" is an implementation of the first step described in Chapter 3. To be able to use statistical debugging techniques to perform fault localization, *SimFL* requires collecting the test

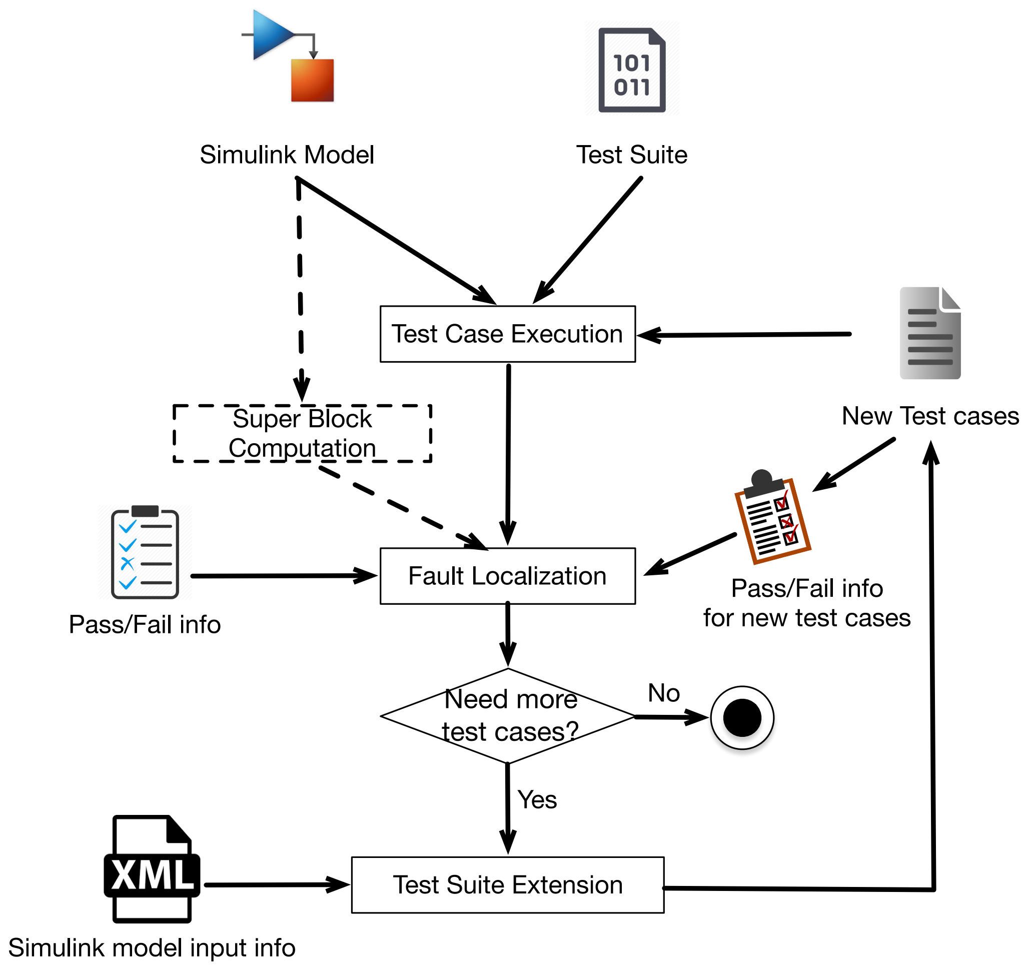


Figure 6.1. Workflow of SimFL

coverage information for each model output and each test case. In this step, *SimFL* takes as input a test suite file, and a (faulty) Simulink model and calls Matlab/Simulink to run the Simulink model with the given test suite. The output of this step is a file containing which decision being made by individual conditional blocks (e.g. Switch) in the model. This decision information can be obtained by analyzing Matlab/Simulink Model Coverage Report Generator data. *SimFL* computes the actual coverage data for conditional blocks and then exports the data into a file in a specific format.

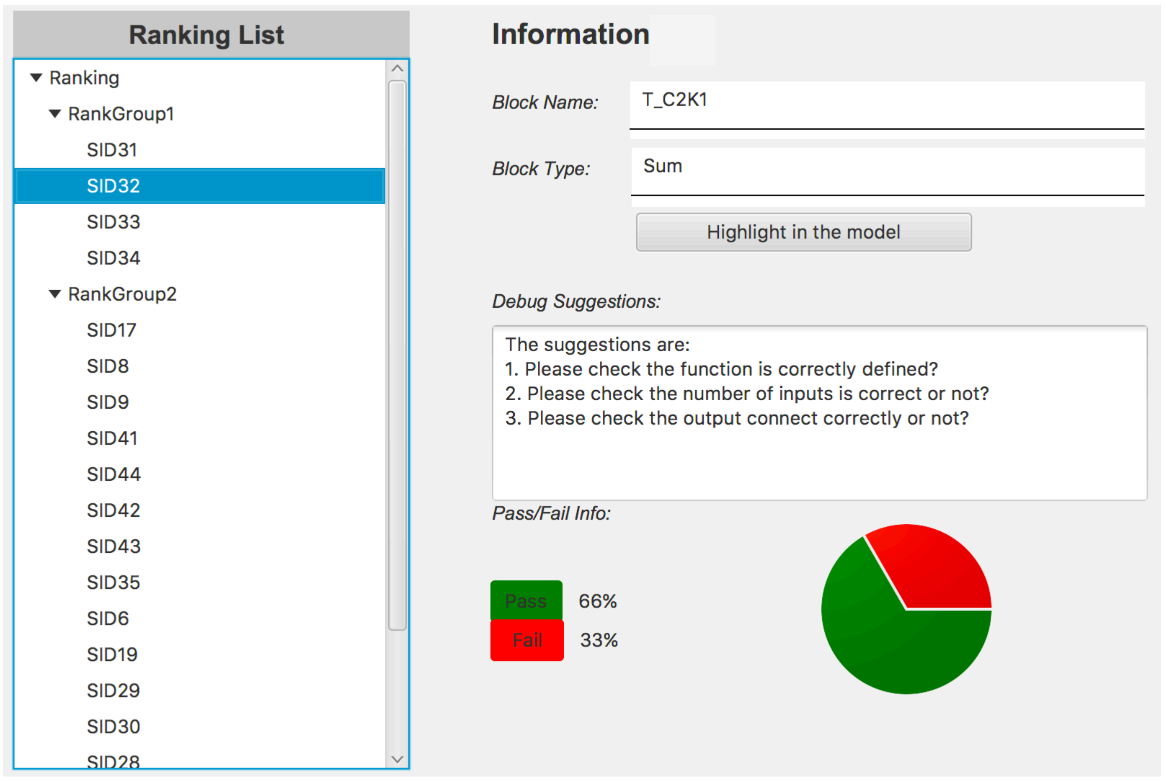
6.1.2 Super Block Computation

The "*Simulink super block*" is a new concept we proposed in Chapter 5 which could help engineers understand the generated rankings and help them decide whether further extending the test suite is likely to be beneficial (as discussed in Section 5.2.2). In brief, a *Simulink super block* is a set of atomic Simulink blocks whose suspiciousness scores cannot be differentiated by adding new test cases.

Computing super block is based on the static analysis of a Simulink model, so it only needs to be computed once if there is no change in the model output list. *SimFL* offers the computation of super blocks as an optional function (marked with dashed line in Figure 6.1). Users can decide by themselves to generate it or not based on their needs. However, *SimFL* detects the existence of the super block information file to decide about structure view of the generated ranked list visualization in the debugging dialogue.

6.1.3 Fault Localiation

Once the test execution information has been obtained, *SimFL* is ready to generate a ranked list of blocks for engineers to debug their Simulink model. In this part, we discuss: (1) Statistical Debugging techniques that we used to generate the ranked list to inspect and (2) The debugging window in *SimFL* and the way *SimFL* connects/interacts with Matlab/Simulink.



Block Name: T_C2K1

Block Type: Sum

Debug Suggestions:

The suggestions are:
1. Please check the function is correctly defined?
2. Please check the number of inputs is correct or not?
3. Please check the output connect correctly or not?

Pass/Fail Info:

Pass	66%
Fail	33%

Figure 6.2. A screenshot of the debugging window

6.1.3.1 Statistical Debugging

Our fault localization approach that has been implemented in *SimFL* is based on the statistical debugging technique described in Chapter 3. *SimFL* provides three alternative statistical formulae: Tarantula [Jones et al., 2002], Ochiai [Abreu et al., 2007], and DStar [Wong et al., 2014]. Users can select any of the formulae to compute the block suspiciousness scores. The output of statistical debugging is a ranked list of Simulink atomic blocks.

We implemented the slicing technique we proposed and explained in Chapter 3 to compute the spectra information. Particularly, based on the coverage information generated in the previous step, *SimFL* prunes and refines the backward static slices to generate the test execution slices that contain precisely the information about the execution of atomic blocks in the model for each output. The other input, the pass/fail information, has to be provided by users.

6.1.3.2 Debugging user interface design and Connection with Matlab/Simulink

After generating a ranked list, *SimFL* presents the ranked list as a tree structure as shown in Figure 6.2. The elements (blocks) listed in the tree are organized based on their suspiciousness score from high to low. The elements are put under the same rank group when they have the same suspiciousness score.

Figure 6.2 shows the user interface design for the debugging step. This window is divided into two parts: on the left part, *SimFL* presents the generated ranked list into a tree structure. The right part shows the related block information of the selected elements in the list, including block name, block type, bug fixing suggestions, and percentage of passed/failed test cases. On the left part, each element in the ranked list represents an atomic block in the Simulink model. In the ranked list, *SimFL* uses Simulink *SID* instead of Block Name as the identification of a block. This is because, in a Simulink model, several blocks that are located in different subsystems could have the same name. Hence, Block Name is not enough to differentiate different blocks. But, Simulink *SID* is unique for each block in a Simulink model.

Whenever an element (SID) in the ranking list is selected, all information related to this block is displayed in the right part of the window. In order to help engineers quickly find the exact location of the block, *SimFL* also provides a button ("Highlight in the model" button in Figure 6.2) to bring engineers back to the corresponding Simulink model. When this button is clicked, the model is opened and the selected element is highlighted in a bright blue color.

6.1.4 Test Suite Extension

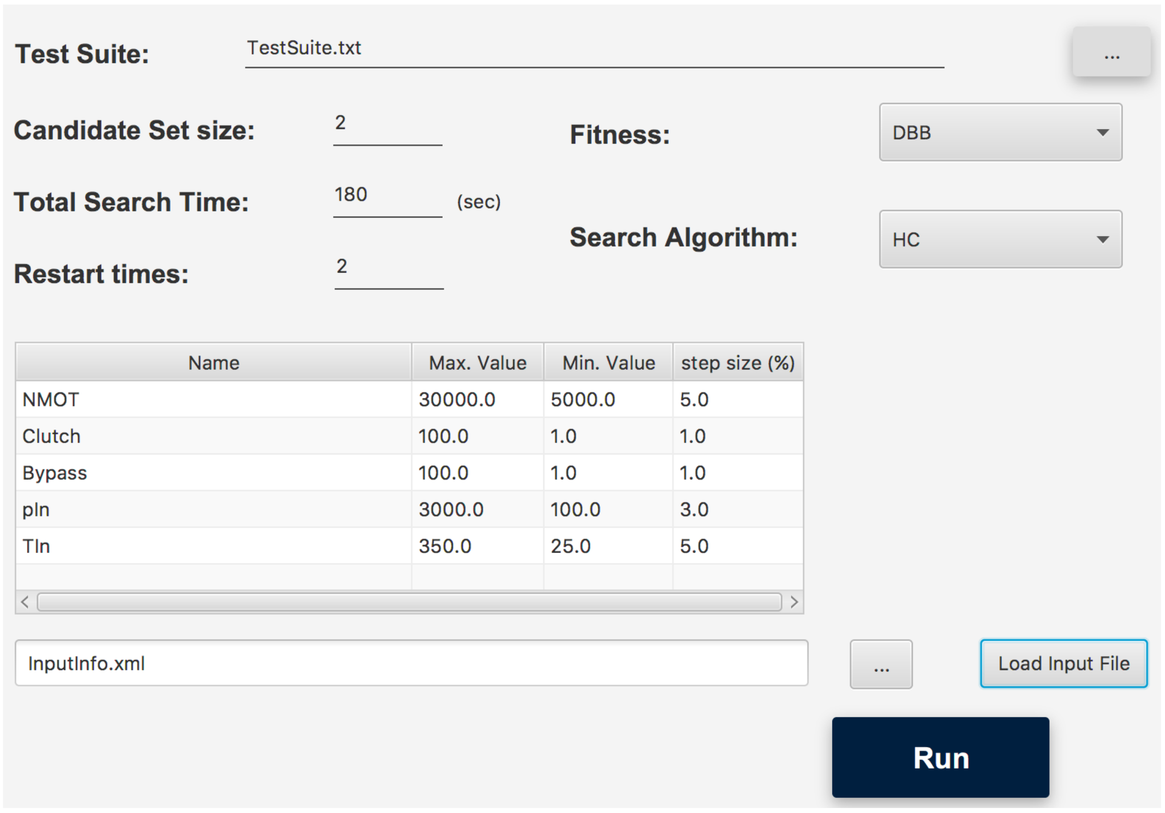
In statistical debugging results, several elements (blocks) may have the same suspiciousness scores, and hence, form coarse-grained rank groups. If engineers inspect blocks using such a ranked list, they may have to inspect a large number of blocks to find out the faulty one(s). To improve the precision of statistical debugging techniques, *SimFL* provides a test suite extension strategy based on search-based techniques to refine the ranking.

Figure 6.3 shows the test case generation setting form where the user can specify the necessary information for *SimFL* to run a search-based test case generation approach (discussed in Chapter 5). In this window, users have to specify the current test suite file, the Simulink model inputs as well as the configuration/selection parameters for running search algorithms. Once all these parameters have been correctly configured, *SimFL* will start generating new test cases. At this stage, Simulink is invoked to execute the test case candidates.

6.2 Implementation

SimFL has been implemented as a standalone application. The pre-requisites of the tool are (i) JDK version later than 1.7, and (ii) Matlab/Simulink version later than 2012. *SimFL* can work on both MacOS and WindowsOS environments.

Figure 6.4 shows the architecture view of *SimFL*. The GUI is implemented in JavaFX. The four functional modules in the core engine part are implemented in Java. We adopt a third-party Java API, matlabcontrol [matlabcontrol, 2010], to setup the communication between the *SimFL* and Mat-



Test Suite: TestSuite.txt

Candidate Set size: 2

Total Search Time: 180 (sec)

Restart times: 2

Fitness: DBB

Search Algorithm: HC

Name	Max. Value	Min. Value	step size (%)
NMOT	30000.0	5000.0	5.0
Clutch	100.0	1.0	1.0
Bypass	100.0	1.0	1.0
pIn	3000.0	100.0	3.0
TIn	350.0	25.0	5.0

InputInfo.xml

Figure 6.3. A screenshot of the test suite extension window

lab/Simulink. The Simulink model executions in both the Execution step and Test Suite Generation step are implemented in Matlab scripts.

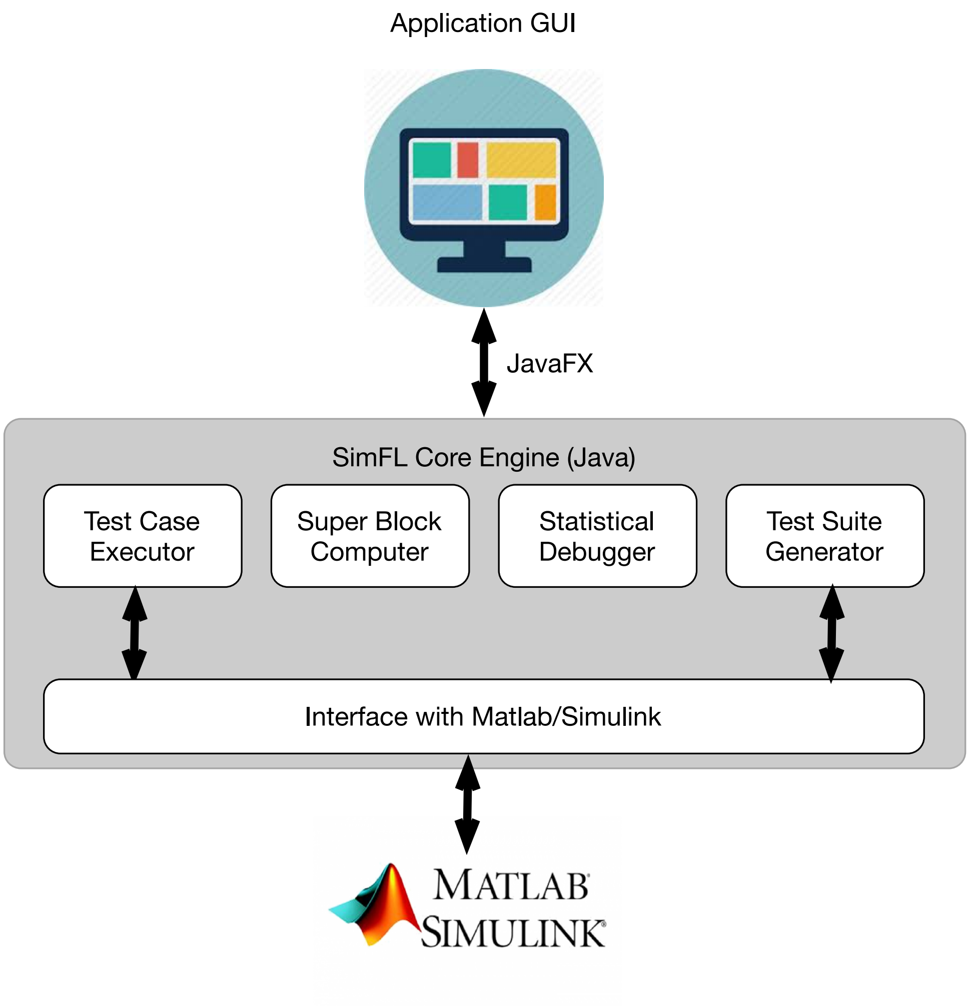


Figure 6.4. *SimFL* architecture view

Chapter 7

Related Work

In this chapter, we focus on comparing our work in this dissertation with the related research and studies. We initially present existing approaches and their suitability to solving the challenges of single fault problem in source code (Section 7.1). Next, we present the existing work on the analysis of Simulink models (Section 7.2). We also review existing work that aims to localize multiple faults in programs (Section 7.3). Section 7.4 provides a review of research as they relate to the test suite generation for improving fault localization result.

7.1 Software Fault Localization for single-fault

Many fault localization techniques have been proposed to localize faults in programs [Abreu et al., 2009b, Abreu et al., 2007, Abreu et al., 2009c, Alves et al., 2011, Arumuga Nainar and Liblit, 2010, Ball et al., 2003, Chen et al., 2002, Chilimbi et al., 2009, Cleve and Zeller, 2000, Cleve and Zeller, 2005, Groce et al., 2004, Hildebrandt and Zeller, 2000, Jones and Harrold, 2005, Jones et al., 2002, Kim et al., 2015, Lei et al., 2012, Liblit et al., 2005, Liu et al., 2005, Lucia et al., 2010, Mayer et al., 2009, Orso et al., 2004, Parnin and Orso, 2011, Parsa et al., 2014, Renieris and Reiss, 2003, Santelices et al., 2009, Tang et al., 2014, Wong et al., 2008, Wong et al., 2014, Xie et al., 2013b, Zhang et al., 2006, Zhang et al., 2003, Zoeteweij et al., 2007]. Statistical debugging is one family of fault localization approaches that has been extensively studied to localize faults in programs [Abreu et al., 2009b, Abreu et al., 2007, Abreu et al., 2009c, Alves et al., 2011, Arumuga Nainar and Liblit, 2010, Chen et al., 2002, Chilimbi et al., 2009, Jones and Harrold, 2005, Jones et al., 2002, Kim et al., 2015, Lei et al., 2012, Liblit et al., 2005, Liu et al., 2005, Lucia et al., 2010, Mayer et al., 2009, Parsa et al., 2014, Renieris and Reiss, 2003, Santelices et al., 2009, Tang et al., 2014, Wong et al., 2014, Xie et al., 2013b, Zoeteweij et al., 2007]. Nevertheless, statistical debugging has not been studied to localize faults in Simulink models. In this work, we propose a statistical debugging technique that takes into account the characteristic of Simulink in order to localize faults in Simulink models.

To identify faults in programs, statistical debugging techniques analyze program spectra and use a statistical formula to measure the likelihood of program elements to be faulty. Different types of program spectra have been analyzed to localize faults, e.g. sequences of statements [Jones and Harrold, 2005, Jones et al., 2002, Renieris and Reiss, 2003, Xie et al., 2013b, Wong et al., 2014], program blocks [Abreu et al., 2009b, Abreu et al., 2007, Lucia et al., 2010, Renieris and Reiss, 2003, Zoeteweij et al., 2007], predicates [Liblit et al., 2005, Liu et al., 2005, Parsa et al., 2014], combination of spectra [Santelices et al., 2009], program path [Chilimbi et al., 2009]. A number of statistical formulas to measure suspiciousness of program elements have also been proposed e.g., *Tarantula* [Jones et al., 2002], *Ochiai* [Abreu et al., 2009b, Abreu et al., 2007], formulas from data mining [Lucia et al., 2010], Naish [Naish et al., 2011], D^* [Wong et al., 2014], formulas generated using genetic programming [Xie et al., 2013b], SOBER [Liu et al., 2005], CBI [Liblit et al., 2005]. In this work, we analyze sequences of (atomic) blocks in Simulink and use existing statistical formulas (i.e., Tarantula, Ochiai, and D^*) to measure the suspiciousness of Simulink (atomic) blocks to be faulty.

Tang et al. [Tang et al., 2014] build a hierarchy of program predicates using hierarchical clustering and uses it to compute the suspiciousness of each predicate. Parsa et al. [Parsa et al., 2014] focus on comparing the effectiveness of ranking results for different code abstractions. Our work uses intermediary model outputs obtained from different subsystems at different hierarchical levels, and focuses on extending test oracles based on these outputs.

The above techniques [Abreu et al., 2009b, Abreu et al., 2007, Chen et al., 2002, Jones and Harrold, 2005, Jones et al., 2002, Liblit et al., 2005, Liu et al., 2005, Lucia et al., 2010, Renieris and Reiss, 2003, Santelices et al., 2009, Xie et al., 2013b, Zoeteweij et al., 2007] localize faults by performing statistical debugging technique only once. Other debugging techniques [Arumuga Nainar and Liblit, 2010, Chilimbi et al., 2009, Zuo et al., 2014] iteratively apply a statistical debugging technique until developers find the root cause of failures. Arumuga et al. [Arumuga Nainar and Liblit, 2010] and Chilimbi et al. [Chilimbi et al., 2009] first instrument selected program elements and apply a statistical debugging technique to obtain the most suspicious program element. Developers then check whether the most suspicious program element is faulty or not. If the suspicious element is not faulty, these techniques extend their instrumentation to other program elements, and a statistical debugging technique is applied again to locate faults. Chilimbi et al. [Chilimbi et al., 2009] search the location of faults by extending their instrumentation to include program elements (i.e., functions) that are highly dependent on the non-suspicious program elements (e.g., functions, branches). A program element is not suspicious if their suspiciousness score is less than a threshold. Arumuga et al. [Arumuga Nainar and Liblit, 2010] extend their instrumentation to include program elements (i.e., predicates) that are nearby to the most suspicious program element (i.e., predicates). Their intuition is that predicates that are nearby to the most suspicious predicate are also suspicious. Instead of extending the instrumentation to include other program elements, Zuo et al. [Zuo et al., 2014] search the location of faults using hierarchical instrumentation. They first instrument functions in a program and use a statistical debugging technique to rank functions. They then instrument predicates of the functions that appear in the top rank and run the statistical debugging technique to locate the faulty predicates. The existing iterative debugging techniques [Arumuga Nainar and Liblit, 2010, Chilimbi et al., 2009, Zuo et al., 2014] focus on extending and improving program instrumentation to reduce memory and time required by the instrumentation. In our work, however, we focus on extending test oracles to improve the accuracy of statistical debugging in localizing faults. Further, our approach does not require engineers to inspect the ranked list first in order to decide whether or not another iteration is needed, since our heuristic automatically predicts whether another iteration of fault localization is needed or not.

Gong et al. [Gong et al., 2012] refine suspiciousness rankings returned by a statistical debugging techniques by using developer feedback (i.e., whether a program element is faulty or not) to adjust the suspiciousness scores of program elements and rerank the program elements. Our approach refines the ranked lists by asking engineers whether some selected intermediary outputs are correct or not, and use this information to narrow down the potential faulty Simulink blocks.

Program slicing has been used to refine ranking results produced by statistical debugging techniques [Abreu et al., 2009c, Alves et al., 2011, Hofer and Wotawa, 2012, Lei et al., 2012, Mao et al., 2014, Mayer et al., 2009]. Alves et al. [Alves et al., 2011] first obtain one spectrum for each test case and apply statistical debugging on these spectra. They then prune the ranking results by removing the statements that are not in the dynamic slicing of incorrect outputs. Our work is different as instead of using dynamic slicing to prune the ranking results, we first obtain one spectrum (test execution slice) per output and per test case, and then apply statistical debugging on all the test execution slices. When multiple outputs are incorrect in a test execution, the techniques proposed by Lei et al. [Lei et al., 2012] and Mao et al. [Mao et al., 2014] rely on the spectra related only to the first failing output and use only those spectra to compute rankings. Hofer et al. [Hofer and Wotawa, 2012] use the spectra from incorrect outputs to generate sets of program elements that can explain failures. In our work, we use spectra from all outputs instead of only one output [Lei et al., 2012, Mao et al., 2014] or only incorrect outputs [Hofer and Wotawa, 2012]. Further, we compute block scores based on the combined spectra related to all the outputs. Note that in our work, we notice that some output may fail for all test executions. That is, considering the spectra only from one output is not sufficient to produce meaningful rankings. Finally, we also provide a heuristic to improve the ranking results by guiding engineers on how to extend test oracles.

Approximate Dynamic Backward Slicing (ADBS) [Mao et al., 2014] bears some similarities to our notion of test execution slice. To clarify the differences between these two, we note that our notion of test execution slice accounts for the blocks that are executed by a test case and affect a specific output generated by that test case. However, ADBS contains program statements executed by a test case and appearing in the backward static slice of a specific output, but these statements do not necessarily affect the output generated by that test case.

Statistical debugging has been previously extended to spreadsheets [Hofer et al., 2014, Hofer et al., 2013, Jannach et al., 2014] and to multi-agent systems [Passos et al., 2015] as well. Similar to our work, in both of these approaches, the notion of spectrum is defined per test case and per output. Specifically, in the spreadsheet fault localization approach, only one test case is used for each spreadsheet, and each execution of that test case for each output amounts to one spectrum [Hofer et al., 2014, Hofer et al., 2013, Jannach et al., 2014]. In the multi-agent system fault localization, the behavior of each agent at each time step is considered to be a spectrum [Passos et al., 2015]. Our work bears some similarities with these two approaches with respect to the notion of spectrum and application of ranking formulas. However, our approach is applied to Simulink models that are drastically different from spreadsheets and multi-agent systems in terms of structure and usage, and further, the computation of Simulink model spectra (test execution slices) is significantly different in our work compared to these approaches.

Gonzales et al. [Gonzalez-Sanchez et al., 2011] and Campos et al. [Campos et al., 2013], respectively, prioritize and generate test cases with the goal of reducing the size and number of ambiguity groups in ranked lists. In our work, we use the maximum size of the top-most ranked ambiguity groups in a ranked list as a heuristic to choose whether to continue expanding test oracles or not. Specifically, our coarseness measure differs from their coarseness measure [Campos et al., 2013, Gonzalez-Sanchez et al., 2011] in that we focus on the size of ambiguity groups that ranked in the top of the list as opposed to the size of ambiguity groups in the entire list.

Statistical debugging assumes developers can find faults by inspecting statements in isolation, while in reality they often need context information to decide if a statement is faulty or not [Parnin and Orso, 2011]. Like existing work, we generate block rankings without including context information. However, Simulink blocks often contain some implicit context information since engineers often label them with terms coming from requirements or architecture. For example, the multiplication block with label `IncrPres` in Figure 3.1 refers to an operation for increasing the pressure of supercharger. This observation suggests that block rankings could be useful to find faults in Simulink.

When researchers propose a new fault localization technique, they typically compare their technique with other techniques by using some known faults (e.g., Siemens suite [Hutchins et al., 1994]) to determine which one is better. Recently, Pearson et al. [Pearson et al., 2017] compared different fault localization techniques (5 from spectrum-based and 5 from mutation-based families) using real faults from different open-source projects. Some of their conclusions are consistent with our conclusions obtained based on experiments on real industrial faulty Simulink models in Chapter 3. Specifically, similar to Pearson et al. [Pearson et al., 2017], we did not observe any significant difference between different fault localization formulas (e.g., Dstar and Tarantula) in our experiments applied to Simulink models based on realistic faults from industry.

7.2 Analysis of Simulink Models

In our work, we relied on model simulations to identify control dependencies between Simulink blocks. Reicherdt and Glesner [Reicherdt and Glesner, 2012] proposed a slicing method for Simulink models where control dependencies are obtained via Simulink Conditional Execution Contexts (CECs) and are used to create static slices based on a set of blocks. In our work, we chose to use model execution information to identify control dependencies and compute slices since the static slicing proposed by Reicherdt et al. [Reicherdt and Glesner, 2012] and Sridhar et al. [Sridhar and Srinivasulu, 2014] based on CECs may provide over approximations that may not be sufficiently precise to determine control dependencies.

Our work relates to the recent work of Schneider [Schneider, 2014] that proposes a technique for tracking the root causes of defects in Simulink. In that technique, engineers identify failures, typically run-time failures, at the level of code generated from Simulink models. The program statement that exhibits the failure is then mapped to a Simulink block, and all the paths leading to that block are collected and assigned weights based on some heuristic. The path with the highest weight is then reported to the engineer as the root cause of the defects. This work focuses on runtime failures (e.g., division by zero), while in our work, we consider a wider range of fault types for Simulink models (see Section 3.4.2). Further, Schneider [Schneider, 2014] does not provide any realistic evaluation of the proposed approach. In particular, the number of blocks that engineers need to eventually inspect is not reported. Finally, the scalability of the approach to large models is not discussed as the number of paths leading to a specific block can be very large for real-world Simulink models.

7.3 Fault Localization for multiple faults

In this section, we compare our work with the existing fault localization techniques that aim to localize multiple faults in programs [Podgurski et al., 2003, Steimann and Frenkel, 2012, Liu and Han, 2006, Jones et al., 2007, Briand et al., 2007, Zheng et al., 2006, Jiang and Su, 2007, Abreu et al., 2009a, Abreu et al., 2009c, Cellier et al., 2011, Arumuga Nainar and Liblit, 2010, Liblit et al., 2005, Röβler et al., 2012].

Several techniques [Podgurski et al., 2003, Steimann and Frenkel, 2012, Liu and Han, 2006, Jones et al., 2007] cluster failures and require engineers to inspect either failures in all the clusters or all rankings obtained from the clusters. Multidimensional scaling has been used for clustering failures based on similarities between execution profiles in [Podgurski et al., 2003] and between statistical rankings in [Liu and Han, 2006]. Jones et al. [Jones et al., 2007] cluster failures using hierarchical clustering and pairwise clustering. They, further, propose a parallel debugging process to inspect all the ranked lists obtained from all the clusters. Steimann et al. [Steimann and Frenkel, 2012] use integer linear programming to cluster failures and generate statistical rankings for the clusters. These approaches have not been evaluated nor adapted to one-at-a-time debugging. Hence, they require engineers to inspect all the clusters (or all the ranked lists) at once, potentially missing the masked faults or wasting effort by identifying the same faults more than once. Further, none of these techniques use both input values and execution slices (traces) as the input for clustering. Our work uses a supervised learning technique applied to heterogeneous input data, and is designed and evaluated for a one-at-a-time debugging process, matching how Simulink models are often debugged in practice.

Instead of generating several rankings, the following techniques generate one ranking. Zheng et al. [Zheng et al., 2006] cluster failing test executions and predicates. Jiang and Su [Jiang and Su, 2007] cluster predicates using a k-mean technique, identify the most predictive predicates, and generate one ranking in terms of a control-flow graph. Abreu et al. [Abreu et al., 2009a, Abreu et al., 2009c] combine statistical debugging with logical reasoning to rank sets of program elements. Brun and Ernst [Brun and Ernst, 2004] build predictor models based on program revisions to produce a subset of program properties that might be faulty. Cellier et al. [Cellier et al., 2011] cluster failures using association rules to obtain a single ranking and propose a mechanism based on formal concept analysis to guide ranking inspection. These techniques aim to find multiple faults using a single ranking. Only the work in [Cellier et al., 2011] provides guidelines on when an inspection can be stopped, but no evaluation is reported. In contrast, our approach aims to identify one ranking in which at least one fault is top-ranked. Further, our approach is iterative, so that faults that are ranked low in the first iterations, can be ranked higher in subsequent iterations. Finally, our work is evaluated using industrial case studies.

Liblit et al. [Liblit et al., 2005, Arumuga Nainar and Liblit, 2010] iteratively re-rank predicates using only the execution traces that do not execute the top-ranked predicates identified in the previous iterations. As they do not re-generate execution traces after fixing faults, the masked faults may remain undetected. In our earlier work [Briand et al., 2007], we have also used decision trees based on input equivalence classes to cluster failures in the context of black-box testing. Our current work uses both execution traces and test inputs for clustering. Further, we assume a one-at-a-time debugging process and select one ranking per iteration, while in [Briand et al., 2007], clusters are combined together to generate a single ranking and the approach is not iterative. Finally, we have applied our technique to industrial Simulink models with multiple faults. None of the above have been applied to Simulink models.

7.4 Test suite generation

Many test generation techniques have been proposed for different purposes e.g., maximizing program coverage ([Fraser and Arcuri, 2013, Harman et al., 2010, Williams et al., 2005, Xie et al., 2005, Sen et al., 2005, Godefroid et al., 2005, Tillmann and De Halleux, 2008, Korel, 1990, Wegener et al., 2001, Baars et al., 2011, Inkumsah and Xie, 2008, Malburg and Fraser, 2011, Tonella, 2004, Ribeiro, 2008, Wappler and Lammermann, 2005]) and revealing faults ([Artzi et al., 2008, Godefroid et al., 2005, Tillmann and De Halleux, 2008, Fraser and Zeller, 2012, DeMilli and Offutt, 1991, Offutt et al., 1999, Jones et al., 1998, Ayari et al., 2007, Evans and Savoia, 2007, Orso and Xie, 2008, Pacheco and Ernst, 2005, Robinson et al., 2011, Alipour et al., 2016, Tang et al., 2016]) for programs, or maximizing structural coverage [Windisch, 2009, Windisch, 2010, Mohalik et al., 2014, Hamon et al., 2008, Satpathy et al., 2008, Sims and DuVarney, 2007, Böhr and Eschbach, 2011, Gadkari et al., 2008, Peranandam et al., 2012, Satpathy et al., 2012] and revealing faults [Matinnejad et al., 2016, Matinnejad et al., 2015b, Zhan and Clark, 2005, Zhan and Clark, 2008, Brillout et al., 2009, Cleaveland et al., 2006, Barnat et al., 2012, Mazzolini et al., 2010, Venkatesh et al., 2012, Holling et al., 2014, Balasubramanian et al., 2011] for Simulink models. Nevertheless, only a few test generation techniques aim to improve fault localization accuracy. These techniques specifically focus on Java/C programs [Rö*β*ler et al., 2012, Baudry et al., 2006, Campos et al., 2013] and on web applications [Artzi et al., 2010]. Our work aims to improve fault localization accuracy for Simulink models by extending an existing test suite with a *small* number of test cases. This is to ensure applicability of our approach in situations where test oracles are developed manually or running test cases is expensive.

One important requirement in our work is that the pass/fail information for each candidate test input is not readily available, and hence, test generation techniques that require such information to improve fault localization [Rö*β*ler et al., 2012, Artzi et al., 2010] are not applicable in our case since these techniques are feasible only when test oracles are automatable. Hence, in our work, we identify the test generation techniques of [Baudry et al., 2006] and [Campos et al., 2013] that satisfy our requirement. Both of these techniques attempt to generate test cases that execute varying subsets of program statements. In particular, Baudry et. al. [Baudry et al., 2006] guide test generation by maximizing the number of Dynamic Basic Blocks (i.e. program elements that are always executed together), and Campos et. al. [Campos et al., 2013] attempt to generate test cases with a balanced number of long and short structural test coverages. In our work, we adapt these two test generation algorithms to Simulink models. In addition, we introduce a new test generation objective that has previously been used for test prioritization [Jiang et al., 2009] and use it to improve fault localization for Simulink models. In contrast to the work of [Baudry et al., 2006, Campos et al., 2013, Jiang et al., 2009], we assess the capabilities of test generation techniques in improving Simulink fault localization when the number of newly generated test cases is small. We, further, combine these techniques with a predictor model that stops test generation when new test cases are not likely to help improve fault localization accuracy.

Most recently, Perez et al. [Perez et al., 2017] noted some possible limitations in the previous metrics ([Campos et al., 2013] and [Baudry et al., 2006]). They proposed a test-suite diagnosability metric by combining these two existing metrics, as well as another measurement defined based on coverage diversity [Jost, 2006]. They evaluated their approaches by implementing the new metric in EVOSUITE [Fraser and Arcuri, 2011]. They showed that their DDU metric needs less effort compared to the Branch-coverage metric. Since DDU does not need the test oracle information, it is possible to adapt DDU into our approach as another test objective.

Le and Lo [Le and Lo, 2013] propose an approach to predict fault localization accuracy based on features extracted from statistical rankings generated by a fixed and specific test suite. Our predictor model instead is built based on features that compare statistical rankings generated by a test suite and its extensions. Moreover, our predictor model is used to help stop test generation and to ensure test suite minimality. Further investigation is required to assess the effectiveness of the features proposed in [Le and Lo, 2013] as a test generation stopping criterion.

Xia et al. [Xia et al., 2016] select a subset of a given test suite such that the fault localization accuracy achieved by the subset is the same as the accuracy achieved by the entire test suite. Similar to our work, they create predictor models based on changes in rankings as new test cases are added to the underlying test suite. However, they build a predictor model for each program element as opposed to our work where we build one predictor model based on the changes in the top-N ranked groups. As discussed earlier, since Simulink atomic blocks in the same super block always have the same rank, creating separate predictors for each individual atomic blocks is too fine-grained and redundant. Furthermore, at each round, in order to select a test case, Xia et al. [Xia et al., 2016] need to compare the spectra of the candidate test case with those of all the remaining test cases. This makes their approach computationally and memory intensive when the test suite from which test cases are selected is large. In our work, however, we extend an initial test suite using a search-based test generation technique guided by objectives that aim to increase test suite diversity without any need to compare the spectra of many test cases.

Chapter 8

Conclusions and Future Work

In this chapter, we revisit the main contributions of this dissertation and discuss some perspectives on potential research directions in this area.

This chapter is organized as follow. Section 8.1 summarizes the contributions of this dissertation. Section 8.2 discusses potential future work.

8.1 Summary

As the most important industrial modeling and simulation language, Simulink has a lot of unique features: Simulink models often have many inputs and output signals, contain hundreds of blocks and lines, and are often hierarchical i.e., including subsystems. Furthermore, subsystems and their connecting lines may form a closed-loop. The complex structure of Simulink models makes debugging highly difficult and time-consuming. In addition, developing test oracles for Simulink models is challenging. Hence, having some effective techniques that are able to automatically reveal, identify and locate faults in Simulink models is beneficial for engineers. In this dissertation, we proposed several fault localization approaches aimed at addressing the challenges of identifying faults in industrial Simulink models, and a test generation algorithm for improving the fault localization for Simulink models.

To conclude, we presented the following contributions in this dissertation:

Chapter 3 presented our fault localization approach for single-fault Simulink models. We proposed SimFL, a fault localization approach for Simulink models by combining statistical debugging and dynamic model slicing. Given a set of outputs in a Simulink model that needs to be debugged, we generate test execution slices, for each test case and output, of the Simulink model. We then apply statistical ranking formulas to the resulting spectra to compute suspiciousness scores for each Simulink model block. In order to further improve fault localization accuracy, we propose iSimFL, an iterative fault localization algorithm. At each iteration, iSimFL increases the set of observable outputs by including outputs at lower hierarchy levels, thus increasing the test oracle cost but offsetting it with significantly more precise fault localization. We utilize a heuristic stopping criterion to avoid unnecessary test oracle extension. We evaluate our work on three industrial Simulink models from Delphi Automotive. Our results show that, on average, SimFL ranks faulty blocks in the top 8.9%

in the list of suspicious blocks. Further, we show that iSimFL significantly improves this percentage down to 4.4% by requiring engineers to observe only an average of five additional outputs at lower hierarchy levels on top of high-level model outputs.

Chapter 4 described our approach for multi-fault Simulink model fault localization. Our approach builds on statistical debugging and is iterative. At each iteration, we identify and resolve one fault and re-test models to focus on localizing faults that might have been masked before. We use decision trees to cluster together failures that satisfy similar (logical) conditions on model blocks or inputs. We then present two alternative selection criteria to choose a cluster that is more likely to yield the best fault localization results among the clusters produced by our decision trees. Engineers are expected to inspect the ranked lists obtained from the selected cluster to identify faults. We evaluate our approach on 240 multi-fault models obtained from three different industrial subjects. We compare our approach with two baselines: (1) Statistical debugging without clustering, and (2) State-of-the-art clustering-based statistical debugging. Our results show that our approach significantly reduces the number of blocks that engineers need to inspect in order to localize all faults, when compared with the two baselines. Furthermore, with our approach, there is less performance degradation than in the baselines when increasing the number of faults in the underlying models.

Chapter 5 presented our test generation approaches that aim at improving fault localization of Simulink models by generating test cases. In this chapter, we identified several test objectives that aim to increase test suite diversity. We used these objectives in a search-based algorithm to generate diversified but small test suites. To further minimize test suite sizes, we developed a prediction model to stop test generation when adding test cases is unlikely to improve fault localization. We evaluated our approach using three industrial subjects. Our results show (1) the selected test objectives are able to significantly improve the accuracy of fault localization for small test suite sizes, and (2) our prediction model is able to maintain almost the same fault localization accuracy while reducing the average number of newly generated test cases by more than half.

Chapter 6 presented our tool providing automated support for our approach and the features we proposed for Simulink model fault localization. The tool enables users to exercise Simulink models for a given test suite, extract and compute test execution information, generate fault localization ranking lists for a selected configuration, view/inspect the ranking lists interactively, and extend a given test suite based on a selected fitness metric. All of these features have been evaluated in the empirical evaluation section from Chapter 3 to Chapter 5.

All of the work presented in this dissertation has been done in collaboration with Delphi Automotive Systems, a world leading automotive part supplier company, based in Luxembourg.

8.2 Future Work

In this dissertation, we focused on fault localization for Simulink models developed in the automotive domain. To better assess the applicability and effectiveness of our approaches, we identified the following topics/areas of interest:

1. In this dissertation, all the approaches target the Simulink models without stateflow. The main reason is that the industrial subjects we got from Delphi did not contain any. One of the future

studies is to extend our approaches to localize faults in the Simulink models with stateflow structures. Some useful work [Sridhar and Srinivasulu, 2014, Matinnejad et al., 2015b] about slicing and testing stateflow has already been conducted and can be used to expand our fault localization approach to stateflows.

2. In this dissertation, all the empirical evaluations are conducted based industrial subjects from our automotive industrial partner. To prove the generalizability of our approaches, it is also worthwhile to consider and evaluate our approaches on other CPS domains, e.g., avionics, communications, and medical systems. Moreover, we believe our approach is not tied to Simulink. Other system modeling notations can also be supported by adapting it, e.g., Labview [National Instruments, 2017].
3. Although we have devised a stand-alone application, *SimFL*, it is still worthwhile to perform user studies with engineers to better understand their needs while debugging, so as to provide additional insights along with the block ranking. This may be particularly interesting for different domains and scenarios based on different practical needs. Furthermore, we believe there are other effective visualization mechanisms to help engineers debug Simulink models.
4. We also intend to study fault localization for evolving Simulink models. A recent study [Rapos and Cordy, 2016] of industrial Simulink models indicates a strong co-evolution relation between changes in models and changes in their corresponding test suites. We plan to investigate how such relations can be used to generate test suites that lead to effective Simulink fault localization, most particularly when models are subject to frequent changes.

List of Papers

Published papers included in this dissertation:

- Bing Liu, Lucia, Shiva Nejati, Lionel C. Briand, Thomas Bruckmann. "Simulink fault localization: an iterative statistical debugging approach." In *Software Testing, Verification and Reliability (STVR)* Journal, pp. 431–459. 2016.
- Bing Liu, Lucia, Shiva Nejati, Lionel C. Briand, Thomas Bruckmann. "Localizing Multiple Faults in Simulink Models." In *Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 146–156. IEEE, 2016.
- Bing Liu, Lucia, Shiva Nejati, Lionel C. Briand. "Improving fault localization for Simulink models using search-based testing and prediction models." In *Proceedings of the 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 359–370. IEEE, 2017.

Unpublished papers included in this dissertation:

- Bing Liu, Shiva Nejati, Lionel C. Briand, Lucia. "Effective Fault Localization of Automotive Simulink Models: Achieving the Trade-Off between Test Oracle Effort and Fault Localization Accuracy." Submitted to the Empirical Software Engineering Journal for publication, 2017.

Bibliography

[Abreu, 2009] Abreu, R. (2009). *Spectrum-based fault localization in embedded software*. PhD thesis, TU Delft, Delft University of Technology.

[Abreu et al., 2009a] Abreu, R., Zoeteweij, P., and Gemund, A. (2009a). Localizing software faults simultaneously. In *Quality Software, 2009. QSIC'09. 9th International Conference on*, pages 367–376. IEEE.

[Abreu et al., 2009b] Abreu, R., Zoeteweij, P., Golsteijn, R., and Van Gemund, A. J. (2009b). A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792.

[Abreu et al., 2007] Abreu, R., Zoeteweij, P., and Van Gemund, A. J. (2007). On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION*, pages 89–98. IEEE.

[Abreu et al., 2009c] Abreu, R., Zoeteweij, P., and Van Gemund, A. J. (2009c). Spectrum-based multiple fault localization. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99. IEEE.

[Agrawal, 1991] Agrawal, H. (1991). *Towards automatic debugging of computer programs*. PhD thesis, Citeseer.

[Alipour et al., 2016] Alipour, M. A., Groce, A., Gopinath, R., and Christi, A. (2016). Generating focused random tests using directed swarm testing. In *Proceedings of the international symposium on Software testing and analysis*. ACM.

[Alpaydin, 2014] Alpaydin, E. (2014). *Introduction to machine learning*. MIT press.

[Alves et al., 2011] Alves, E., Gligoric, M., Jagannath, V., and d'Amorim, M. (2011). Fault-localization using dynamic slicing and change impact analysis. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 520–523. IEEE Computer Society.

[Artzi et al., 2010] Artzi, S., Dolby, J., Tip, F., and Pistoia, M. (2010). Directed test generation for effective fault localization. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 49–60. ACM.

[Artzi et al., 2008] Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A., and Ernst, M. D. (2008). Finding bugs in dynamic web applications. In *Proceedings of the international symposium on Software testing and analysis*, pages 261–272. ACM.

[Arumuga Nainar and Liblit, 2010] Arumuga Nainar, P. and Liblit, B. (2010). Adaptive bug isolation. In *Proceedings of the 32nd International Conference on Software Engineering*, pages 255–264. ACM.

[Ayari et al., 2007] Ayari, K., Bouktif, S., and Antoniol, G. (2007). Automatic mutation test input data generation via ant colony. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1074–1081. ACM.

[Baars et al., 2011] Baars, A., Harman, M., Hassoun, Y., Lakhotia, K., McMinn, P., Tonella, P., and Vos, T. (2011). Symbolic search-based testing. In *Proceeding of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 53–62. IEEE.

[Balasubramanian et al., 2011] Balasubramanian, D., Păsăreanu, C. S., Whalen, M. W., Karsai, G., and Lowry, M. (2011). Polyglot: modeling and analysis for multiple statechart formalisms. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 45–55. ACM.

[Ball et al., 2003] Ball, T., Naik, M., and Rajamani, S. K. (2003). From symptom to cause: localizing errors in counterexample traces. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 97–105.

[Barnat et al., 2012] Barnat, J., Brim, L., Beran, J., Oliveira, Í. R., et al. (2012). Executing model checking counterexamples in simulink. In *Sixth International Symposium on Theoretical Aspects of Software Engineering*, pages 245–248. IEEE.

[Barr et al., 2015] Barr, E. T., Harman, M., McMinn, P., Shahbaz, M., and Yoo, S. (2015). The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525.

[Baudry et al., 2006] Baudry, B., Fleurey, F., and Le Traon, Y. (2006). Improving test suites for efficient fault localization. In *Proceedings of the 28th international conference on Software engineering*, pages 82–91. ACM.

[Ben Abdessalem et al., 2016] Ben Abdessalem, R., Nejati, S., Briand, L. C., and Stifter, T. (2016). Testing advanced driver assistance systems using multi-objective search and neural networks. In *Proceedings of the 31st International Conference on Automated Software Engineering*, pages 63–74. ACM.

[Böhr and Eschbach, 2011] Böhr, F. and Eschbach, R. (2011). Simotest: A tool for automated testing of hybrid real-time simulink models. In *Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference on*, pages 1–4. IEEE.

[Briand et al., 2007] Briand, L. C., Labiche, Y., and Liu, X. (2007). Using machine learning to support debugging with tarantula. In *Software Reliability, 2007. ISSRE'07. The 18th IEEE International Symposium on*, pages 137–146. IEEE.

[Brillout et al., 2009] Brillout, A., He, N., Mazzucchi, M., Kroening, D., Purandare, M., Rümmer, P., and Weissenbacher, G. (2009). Mutation-based test case generation for simulink models. In *International Symposium on Formal Methods for Components and Objects*, pages 208–227. Springer.

[Brun and Ernst, 2004] Brun, Y. and Ernst, M. D. (2004). Finding latent code errors via machine learning over program executions. In *Proceedings of the 26th International Conference on Software Engineering*, pages 480–490. IEEE Computer Society.

[Campos et al., 2013] Campos, J., Abreu, R., Fraser, G., and d'Amorim, M. (2013). Entropy-based test generation for improved fault localization. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 257–267. IEEE.

[Cellier et al., 2011] Cellier, P., Ducassé, M., Ferré, S., and Ridoux, O. (2011). Multiple fault localization with data mining. In *SEKE*, pages 238–243.

[Chaturvedi, 2009] Chaturvedi, D. K. (2009). *Modeling and simulation of systems using MATLAB and Simulink*. CRC Press.

[Chen et al., 2002] Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E. (2002). Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 595–604. IEEE.

[Chen et al., 2005] Chen, T. Y., Leung, H., and Mak, I. (2005). Adaptive random testing. In *Advances in Computer Science-ASIAN 2004. Higher-Level Decision Making*, pages 320–329. Springer.

[Chilimbi et al., 2009] Chilimbi, T. M., Liblit, B., Mehra, K., Nori, A. V., and Vaswani, K. (2009). Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering*, pages 34–44. IEEE.

[Cleaveland et al., 2006] Cleaveland, R., Smolka, S. A., and Sims, S. T. (2006). An instrumentation-based approach to controller model validation. In *Automotive Software Workshop*, pages 84–97. Springer.

[Cleve and Zeller, 2000] Cleve, H. and Zeller, A. (2000). Finding failure causes through automated testing. *arXiv preprint cs/0012009*.

[Cleve and Zeller, 2005] Cleve, H. and Zeller, A. (2005). Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, pages 342–351. ACM.

[Delphi Automotive LLP, 2017] Delphi Automotive LLP (2017). Delphi luxembourg. http://www.delphi.com/about/locations/luxembourg.

[DeMilli and Offutt, 1991] DeMilli, R. and Offutt, A. J. (1991). Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910.

[Domingos, 2012] Domingos, P. (2012). A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87.

[Evans and Savoia, 2007] Evans, R. B. and Savoia, A. (2007). Differential testing: a new approach to change detection. In *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, pages 549–552. ACM.

[Fagin et al., 2003] Fagin, R., Kumar, R., and Sivakumar, D. (2003). Comparing top k lists. *SIAM Journal on Discrete Mathematics*, 17(1):134–160.

[Fraser and Arcuri, 2011] Fraser, G. and Arcuri, A. (2011). Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419. ACM.

[Fraser and Arcuri, 2013] Fraser, G. and Arcuri, A. (2013). Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291.

[Fraser and Zeller, 2012] Fraser, G. and Zeller, A. (2012). Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292.

[Gadkari et al., 2008] Gadkari, A. A., Yeolekar, A., Suresh, J., Ramesh, S., Mohalik, S., and Shashidhar, K. (2008). Automotgen: Automatic model oriented test generator for embedded control systems. In *International Conference on Computer Aided Verification*, pages 204–208. Springer.

[Godefroid et al., 2005] Godefroid, P., Klarlund, N., and Sen, K. (2005). Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM.

[Gong et al., 2012] Gong, L., Lo, D., Jiang, L., and Zhang, H. (2012). Interactive fault localization leveraging simple user feedback. In *Proceedings of the 28th IEEE International Conference on Software Maintenance*, pages 67–76. IEEE.

[Gonzalez-Sanchez et al., 2011] Gonzalez-Sanchez, A., Abreu, R., Gross, H.-G., and Van Gemund, A. J. (2011). Prioritizing tests for fault localization through ambiguity group reduction. In *Proceedings of the 26th International Conference on Automated Software Engineering*, pages 83–92. IEEE.

[Groce et al., 2004] Groce, A., Kroening, D., and Lerda, F. (2004). Understanding counterexamples with explain. In *International Conference on Computer Aided Verification*, pages 453–456. Springer.

[Hamon et al., 2008] Hamon, G., Dutertre, B., Erkok, L., Matthews, J., Sheridan, D., Cok, D., Rushby, J., Bokor, P., Shukla, S., Pataricza, A., et al. (2008). Simulink design verifier-applying automated formal methods to simulink and stateflow. In *Third Workshop on Automated Formal Methods*.

[Harman et al., 2010] Harman, M., Kim, S. G., Lakhotia, K., McMinn, P., and Yoo, S. (2010). Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In *Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 182–191. IEEE.

[Hildebrandt and Zeller, 2000] Hildebrandt, R. and Zeller, A. (2000). Simplifying failure-inducing input. In *ACM SIGSOFT Software Engineering Notes*, volume 25, pages 135–145. ACM.

[Hofer et al., 2014] Hofer, B., Perez, A., Abreu, R., and Wotawa, F. (2014). On the empirical evaluation of similarity coefficients for spreadsheets fault localization. *Automated Software Engineering*, 22(1):47–74.

[Hofer et al., 2013] Hofer, B., Riboira, A., Wotawa, F., Abreu, R., and Getzner, E. (2013). On the empirical evaluation of fault localization techniques for spreadsheets. In *Fundamental Approaches to Software Engineering*, pages 68–82. Springer.

[Hofer and Wotawa, 2012] Hofer, B. and Wotawa, F. (2012). Spectrum enhanced dynamic slicing for better fault localization. In *Proceedings of the 20th European Conference on Artificial Intelligence*, pages 420–425.

[Holling et al., 2014] Holling, D., Pretschner, A., and Gemmar, M. (2014). 8cage: lightweight fault-based test generation for simulink. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 859–862. ACM.

[Hsu et al., 2008] Hsu, H.-Y., Jones, J. A., and Orso, A. (2008). Rapid: Identifying bug signatures to support debugging activities. In *Proceedings of the 23rd International Conference on Automated Software Engineering*, pages 439–442. IEEE Computer Society.

[Hutchins et al., 1994] Hutchins, M., Foster, H., Goradia, T., and Ostrand, T. (1994). Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria. In *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on*, pages 191–200. IEEE.

[Inkumsah and Xie, 2008] Inkumsah, K. and Xie, T. (2008). Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proceeding of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 297–306. IEEE.

[Jaccard, 1901] Jaccard, P. (1901). *Etude comparative de la distribution florale dans une portion des Alpes et du Jura*. Impr. Corbaz.

[Jannach et al., 2014] Jannach, D., Schmitz, T., Hofer, B., and Wotawa, F. (2014). Avoiding, finding and fixing spreadsheet errors–a survey of automated approaches for spreadsheet qa. *Journal of Systems and Software*, 94:129–150.

[Jiang et al., 2009] Jiang, B., Zhang, Z., Chan, W. K., and Tse, T. (2009). Adaptive random test case prioritization. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 233–244. IEEE.

[Jiang and Su, 2007] Jiang, L. and Su, Z. (2007). Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 184–193. ACM.

[Jones et al., 1998] Jones, B. F., Eyres, D. E., and Sthamer, H.-H. (1998). A strategy for using genetic algorithms to automate branch and fault-based testing. *The Computer Journal*, 41(2):98–107.

[Jones et al., 2007] Jones, J. A., Bowring, J. F., and Harrold, M. J. (2007). Debugging in parallel. In *Proceedings of the international symposium on Software testing and analysis*, pages 16–26. ACM.

[Jones and Harrold, 2005] Jones, J. A. and Harrold, M. J. (2005). Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 273–282.

[Jones et al., 2002] Jones, J. A., Harrold, M. J., and Stasko, J. (2002). Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, pages 467–477.

[Jost, 2006] Jost, L. (2006). Entropy and diversity. *Oikos*, 113(2):363–375.

[Kendall, 1938] Kendall, M. G. (1938). A new measure of rank correlation. *Biometrika*, 30:81–93.

[Kim et al., 2015] Kim, J., Park, J., and Lee, E. (2015). A new hybrid algorithm for software fault localization. In *Proceedings of the 9th International Conference on Ubiquitous Information Management and Communication*, page 50. ACM.

[Korel, 1990] Korel, B. (1990). Automated software test data generation. *IEEE Transactions on software engineering*, 16(8):870–879.

[Le and Lo, 2013] Le, T.-D. B. and Lo, D. (2013). Will fault localization work for these failures? an automated approach to predict effectiveness of fault localization tools. In *Proceeding of the 29th IEEE International Conference on Software Maintenance*, pages 310–319. IEEE.

[Le et al., 2014] Le, T.-D. B., Lo, D., and Thung, F. (2014). Should i follow this fault localization tool?s output? *Empirical Software Engineering*, pages 1–38.

[Lee, 2008] Lee, E. A. (2008). Cyber physical systems: Design challenges. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 363–369. IEEE.

[Lei et al., 2012] Lei, Y., Mao, X., Dai, Z., and Wang, C. (2012). Effective statistical fault localization using program slices. In *Proceedings of the 36th Annual Computer Software and Applications Conference*, pages 1–10. IEEE.

[Liblit et al., 2005] Liblit, B., Naik, M., Zheng, A. X., Aiken, A., and Jordan, M. I. (2005). Scalable statistical bug isolation. *ACM SIGPLAN Notices*, 40(6):15–26.

[Liu, 2017] Liu, B. (2017). experiment related. `https://github.com/Avartar/TCGenForFL/`.

[Liu et al., 2017] Liu, B., Lucia, L., Nejati, S., and Briand, L. (2017). Improving fault localization for simulink models using search-based testing and prediction models. In *24th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2017).*

[Liu et al., 2016a] Liu, B., Nejati, S., Briand, L., Bruckmann, T., et al. (2016a). Localizing multiple faults in simulink models. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 146–156. IEEE.

[Liu et al., 2016b] Liu, B., Nejati, S., Briand, L. C., and Bruckmann, T. (2016b). Simulink fault localization: an iterative statistical debugging approach. *Software Testing, Verification and Reliability*, 26(6):431–459.

[Liu and Han, 2006] Liu, C. and Han, J. (2006). Failure proximity: a fault localization-based approach. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 46–56. ACM.

[Liu et al., 2005] Liu, C., Yan, X., Fei, L., Han, J., and Midkiff, S. P. (2005). Sober: statistical model-based bug localization. *ACM SIGSOFT Software Engineering Notes*, 30(5):286–295.

[Lucia et al., 2010] Lucia, Lo, D., Jiang, L., and Budi, A. (2010). Comprehensive evaluation of association measures for fault localization. In *Proceedings of the International Conference on Software Maintenance*, pages 1–10. IEEE.

[Lucia et al., 2014] Lucia, Lo, D., and Xia, X. (2014). Fusion fault localizers. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 127–138.

[Luke, 2015] Luke, S. (2015). Essentials of metaheuristicsl.

[Malburg and Fraser, 2011] Malburg, J. and Fraser, G. (2011). Combining search-based and constraint-based testing. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 436–439. IEEE Computer Society.

[Mao et al., 2014] Mao, X., Lei, Y., Dai, Z., Qi, Y., and Wang, C. (2014). Slice-based statistical fault localization. *Journal of Systems and Software*, 89:51–62.

[MathWorks, a] MathWorks. Simulink Examples. `http://nl.mathworks.com/help/simulink/examples.html`.

[MathWorks, b] MathWorks. Stateflow. `http://www.mathworks.nl/products/stateflow/`.

[Mathworks, 2015] Mathworks (2015). Simulink. https://en.wikipedia.org/wiki/Simulink.

[Matinnejad et al., 2015a] Matinnejad, R., Nejati, S., Briand, L., Bruckmann, T., and Poull, C. (2015a). Search-based automated testing of continuous controllers: Framework, tool support, and case studies. *Information and Software Technology*, 57:705–722.

[Matinnejad et al., 2015b] Matinnejad, R., Nejati, S., Briand, L. C., and Bruckmann, T. (2015b). Effective test suites for mixed discrete-continuous stateflow controllers. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, pages 84–95. ACM.

[Matinnejad et al., 2016] Matinnejad, R., Nejati, S., Briand, L. C., and Bruckmann, T. (2016). Automated test suite generation for time-continuous simulink models. In *proceedings of the 38th International Conference on Software Engineering*, pages 595–606. ACM.

[matlabcontrol, 2010] matlabcontrol (2010). matlabcontrol. `https://code.google.com/archive/p/matlabcontrol/`.

[Mayer et al., 2009] Mayer, W., Abreu, R., Stumptner, M., van Gemund, A., et al. (2009). Prioritising model-based debugging diagnostic reports. In *Proceedings of the International Workshop on Principles of Diagnosis*, pages 127–134.

[Mazzolini et al., 2010] Mazzolini, M., Brusaferri, A., and Carpanzano, E. (2010). Model-checking based verification approach for advanced industrial automation solutions. In *Proceeding of the IEEE Conference on Emerging Technologies and Factory Automation*, pages 1–8. IEEE.

[Mohalik et al., 2014] Mohalik, S., Gadkari, A. A., Yeolekar, A., Shashidhar, K., and Ramesh, S. (2014). Automatic test case generation from simulink/stateflow models using model checking. *Software Testing, Verification and Reliability*, 24(2):155–180.

[Naish et al., 2011] Naish, L., Lee, H. J., and Ramamohanarao, K. (2011). A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)*, 20(3):11.

[National Institute of Standards and Technology U.S. Department of Commerce, 2017] National Institute of Standards and Technology U.S. Department of Commerce (2017). Nist. https://www.nist.gov/.

[National Instruments, 2017] National Instruments (2017). Labview. https://http://www.ni.com/.

[Offutt et al., 1999] Offutt, A. J., Jin, Z., and Pan, J. (1999). The dynamic domain reduction procedure for test data generation. *Software-Practice and Experience*, 29(2):167–93.

[Olshen et al., 1984] Olshen, L., Stone, C. J., et al. (1984). Classification and regression trees. *Wadsworth International Group*, 93(99):101.

[Orso et al., 2003] Orso, A., Jones, J., and Harrold, M. J. (2003). Visualization of program-execution data for deployed software. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 67–ff. ACM.

[Orso et al., 2004] Orso, A., Jones, J. A., Harrold, M. J., and Stasko, J. T. (2004). Gammatella: Visualization of program-execution data for deployed software. In *Proceedings of the 26th International Conference on Software Engineering*, pages 699–700.

[Orso and Xie, 2008] Orso, A. and Xie, T. (2008). Bert: Behavioral regression testing. In *Proceedings of the 2008 international workshop on dynamic analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, pages 36–42. ACM.

[Pacheco and Ernst, 2005] Pacheco, C. and Ernst, M. D. (2005). Eclat: Automatic generation and classification of test inputs. In *European Conference on Object-Oriented Programming*, pages 504–527. Springer.

[Parnin and Orso, 2011] Parnin, C. and Orso, A. (2011). Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 international symposium on software testing and analysis*, pages 199–209. ACM.

[Parsa et al., 2014] Parsa, S., Vahidi-Asl, M., and Asadi-Aghbolaghi, M. (2014). Hierarchy-debug: a scalable statistical technique for fault localization. *Software Quality Journal*, 22(3):427–466.

[Passos et al., 2015] Passos, L. S., Abreu, R., and Rossetti, R. J. (2015). Spectrum-based fault localisation for multi-agent systems. In *Proceedings of the 24th International Conference on Artificial Intelligence*, pages 1134–1140. AAAI Press.

[Pearson et al., 2017] Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M. D., Pang, D., and Keller, B. (2017). Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*, pages 609–620. IEEE Press.

[Peranandam et al., 2012] Peranandam, P., Raviram, S., Satpathy, M., Yeolekar, A., Gadkari, A., and Ramesh, S. (2012). An integrated test generation tool for enhanced coverage of simulink/stateflow models. In *Design, Automation & Test in Europe Conference & Exhibition*, pages 308–311. IEEE.

[Perez et al., 2017] Perez, A., Abreu, R., and van Deursen, A. (2017). A test-suite diagnosability metric for spectrum-based fault localization approaches. In *Proceedings of the 39th International Conference on Software Engineering*, pages 654–664. IEEE Press.

[Podgurski et al., 2003] Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J., and Wang, B. (2003). Automated support for classifying software failure reports. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 465–475. IEEE.

[Rapos and Cordy, 2016] Rapos, E. J. and Cordy, J. R. (2016). Examining the co-evolution relationship between simulink models and their test cases. In *Proceedings of the 8th International Workshop on Modeling in Software Engineering*, pages 34–40. ACM.

[Reicherdt and Glesner, 2012] Reicherdt, R. and Glesner, S. (2012). Slicing matlab simulink models. In *Proceedings of the 34th International Conference on Software Engineering*, pages 551–561. IEEE Press.

[Renieris and Reiss, 2003] Renieris, M. and Reiss, S. P. (2003). Fault localization with nearest neighbor queries. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pages 30–39.

[Ribeiro, 2008] Ribeiro, J. C. B. (2008). Search-based test case generation for object-oriented java software using strongly-typed genetic programming. In *Proceedings of the 10th annual conference companion on Genetic and evolutionary computation*, pages 1819–1822. ACM.

[Röβler et al., 2012] Röβler, J., Fraser, G., Zeller, A., and Orso, A. (2012). Isolating failure causes through test case generation. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 309–319. ACM.

[Robinson et al., 2011] Robinson, B., Ernst, M. D., Perkins, J. H., Augustine, V., and Li, N. (2011). Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 23–32. IEEE Computer Society.

[Santelices et al., 2009] Santelices, R., Jones, J. A., Yu, Y., and Harrold, M. J. (2009). Lightweight fault-localization using multiple coverage types. In *Proceedings of the 31st International Conference on Software Engineering*, pages 56–66. IEEE.

[Satpathy et al., 2012] Satpathy, M., Yeolekar, A., Peranandam, P., and Ramesh, S. (2012). Efficient coverage of parallel and hierarchical stateflow models for test case generation. *Software Testing, Verification and Reliability*, 22(7):457–479.

[Satpathy et al., 2008] Satpathy, M., Yeolekar, A., and Ramesh, S. (2008). Randomized directed testing (redirect) for simulink/stateflow models. In *Proceedings of the 8th ACM international conference on Embedded software*, pages 217–226. ACM.

[Schneider, 2014] Schneider, J. (2014). Tracking down root causes of defects in simulink models. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 599–604.

[Sen et al., 2005] Sen, K., Marinov, D., and Agha, G. (2005). Cute: a concolic unit testing engine for c. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 263–272. ACM.

[Shalev-Shwartz and Ben-David, 2014] Shalev-Shwartz, S. and Ben-David, S. (2014). *Understanding machine learning: From theory to algorithms*. Cambridge university press.

[Sims and DuVarney, 2007] Sims, S. and DuVarney, D. C. (2007). Experience report: the reactis validation tool. In *ACM SIGPLAN Notices*, volume 42, pages 137–140. ACM.

[Sridhar and Srinivasulu, 2014] Sridhar, A. and Srinivasulu, D. (2014). Slicing matlab simulink/stateflow models. In *Intelligent Computing, Networking, and Informatics*, pages 737–743. Springer.

[Steimann and Frenkel, 2012] Steimann, F. and Frenkel, M. (2012). Improving coverage-based localization of multiple faults using algorithms from integer linear programming. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 121–130. IEEE.

[Tang et al., 2014] Tang, C. M., Chan, W. K., and Yu, Y.-T. (2014). Extending the theoretical fault localization effectiveness hierarchy with empirical results at different code abstraction levels. In *Proceedings of the 38th Annual Computer Software and Applications Conference*, pages 161–170. IEEE.

[Tang et al., 2016] Tang, H., Wu, G., Wei, J., and Zhong, H. (2016). Generating test cases to expose concurrency bugs in android applications. In *Proceedings of the 31st IEEE/ACM international Conference on Automated software engineering*. ACM.

[Thums and Quante, 2012] Thums, A. and Quante, J. (2012). Reengineering embedded automotive software. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 493–502. IEEE.

[Tillmann and De Halleux, 2008] Tillmann, N. and De Halleux, J. (2008). Pex–white box test generation for. net. In *International conference on tests and proofs*, pages 134–153. Springer.

[Tonella, 2004] Tonella, P. (2004). Evolutionary testing of classes. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 119–128. ACM.

[TRICENTIS, 2017] TRICENTIS (2017). Software fail watch: 2016 in review. https://www.tricentis.com/resource-assets/software-fail-watch-2016.

[Vargha and Delaney, 2000] Vargha, A. and Delaney, H. D. (2000). A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132.

[Varrette et al., 2014] Varrette, S., Bouvry, P., Cartiaux, H., and Georgatos, F. (2014). Management of an academic hpc cluster: The ul experience. In *Proc. of the Intl. Conf. on High Performance Computing & Simulation*, pages 959–967, Bologna, Italy. IEEE.

[Venkatesh et al., 2012] Venkatesh, R., Shrotri, U., Darke, P., and Bokil, P. (2012). Test generation for large automotive models. In *Proceeding of the IEEE International Conference on Industrial Technology*, pages 662–667. IEEE.

[Wang et al., 2009] Wang, X., Cheung, S.-C., Chan, W. K., and Zhang, Z. (2009). Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *Proceedings of the 31st International Conference on Software Engineering*, pages 45–55. IEEE Computer Society.

[Wappler and Lammermann, 2005] Wappler, S. and Lammermann, F. (2005). Using evolutionary algorithms for the unit testing of object-oriented software. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1053–1060. ACM.

[Wegener et al., 2001] Wegener, J., Baresel, A., and Sthamer, H. (2001). Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854.

[Wikipedia, 2017a] Wikipedia (2017a). Debugging. https://en.wikipedia.org/wiki/Debugging.

[Wikipedia, 2017b] Wikipedia (2017b). Machine learning. https://en.wikipedia.org/wiki/Machine-learning.

[Wilcoxon, 1945] Wilcoxon, F. (1945). Individual comparisons by ranking methods. *Biometrics bulletin*, pages 80–83.

[Williams et al., 2005] Williams, N., Marre, B., Mouy, P., and Roger, M. (2005). Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *European Dependable Computing Conference*, pages 281–292. Springer.

[Windisch, 2009] Windisch, A. (2009). Search-based testing of complex simulink models containing stateflow diagrams. In *Proceeding of the 31st International Conference on Software Engineering-Companion*, pages 395–398. IEEE.

[Windisch, 2010] Windisch, A. (2010). Search-based test data generation from stateflow statecharts. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 1349–1356. ACM.

[Wong et al., 2014] Wong, E., Debroy, V., Gao, R., and Li, Y. (2014). The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308.

[Wong et al., 2008] Wong, E., Wei, T., Qi, Y., and Zhao, L. (2008). A crosstab-based statistical method for effective fault localization. In *Proceedings of the International Conference on Software Testing, Verification, and Validation*, pages 42–51.

[Wong et al., 2016] Wong, W. E., Gao, R., Li, Y., Abreu, R., and Wotawa, F. (2016). A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740.

[Xia et al., 2016] Xia, X., Gong, L., Le, T.-D. B., Lo, D., Jiang, L., and Zhang, H. (2016). Diversity maximization speedup for localizing faults in single-fault and multi-fault programs. *Automated Software Engineering*, 23(1):43–75.

[Xie et al., 2005] Xie, T., Marinov, D., Schulte, W., and Notkin, D. (2005). Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–381. Springer.

[Xie et al., 2013a] Xie, X., Chen, T. Y., Kuo, F.-C., and Xu, B. (2013a). A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4):31.

[Xie et al., 2013b] Xie, X., Kuo, F.-C., Chen, T. Y., Yoo, S., and Harman, M. (2013b). Provably optimal and human-competitive results in sbse for spectrum based fault localisation. In *Search Based Software Engineering*, pages 224–238. Springer.

[Zander et al., 2011] Zander, J., Schieferdecker, I., and Mosterman, P. J. (2011). *Model-based testing for embedded systems*. CRC press.

[Zhan and Clark, 2005] Zhan, Y. and Clark, J. A. (2005). Search-based mutation testing for simulink models. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1061–1068. ACM.

[Zhan and Clark, 2008] Zhan, Y. and Clark, J. A. (2008). A search-based framework for automatic testing of matlab/simulink models. *Journal of Systems and Software*, 81(2):262–285.

[Zhang et al., 2006] Zhang, X., Gupta, N., and Gupta, R. (2006). Pruning dynamic slices with confidence. In *ACM SIGPLAN Notices*, volume 41, pages 169–180. ACM.

[Zhang et al., 2003] Zhang, X., Gupta, R., and Zhang, Y. (2003). Precise dynamic slicing algorithms. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 319–329. IEEE.

[Zheng et al., 2006] Zheng, A. X., Jordan, M. I., Liblit, B., Naik, M., and Aiken, A. (2006). Statistical debugging: simultaneous identification of multiple bugs. In *Proceedings of the 23rd international conference on Machine learning*, pages 1105–1112. ACM.

[Zoeteweij et al., 2007] Zoeteweij, P., Abreu, R., Golsteijn, R., and Van Gemund, A. J. (2007). Diagnosis of embedded software using program spectra. In *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, pages 213–220. IEEE.

[Zuo et al., 2014] Zuo, Z., Khoo, S.-C., and Sun, C. (2014). Efficient statistical debugging via hierarchical instrumentation. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 457–460. ACM.