securityandtrust.lu

Software Verification and Validation Laboratory: Simulink Fault Localization: an Iterative Statistical Debugging Approach

Bing Liu, Lucia, Shiva Nejati, and Lionel Briand

Interdisciplinary Centre for Security, Reliability and Trust

University of Luxembourg

Thomas Bruckmann

Delphi Automotive Systems, Luxembourg

TR-SnT-2015-8

ISBN: 978-2-87971-146-1

September 29, 2015

Version 1.1

UNIVERSITÉ DU LUXEMBOURG

Simulink Fault Localization: an Iterative Statistical Debugging Approach

Bing Liu Lucia Shiva Nejati Lionel Briand
Thomas Bruckmann

October 20, 2015

Abstract

Debugging Simulink models presents a significant challenge in the embedded industry. In this work, we propose *SimFL*, a fault localization approach for Simulink models by combining statistical debugging and dynamic model slicing. Simulink models, being visual and hierarchical, have multiple outputs at different hierarchy levels. Given a set of outputs to observe for localizing faults, we generate test execution slices, for each test case and output, of the Simulink model. 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.

1 Introduction

The embedded software industry increasingly relies on Simulink to develop software [31, 35, 36, 37]. Simulink is a data-flow-based block diagram language for the modeling, simulation, and development of embedded software. The Simulink language, being supported by advanced automated code generators, has become a prevalent 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. Further, Simulink model testing is more likely to help with fault finding compared to testing code as Simulink models are more abstract and more informative for engineers. Given the importance of testing and debugging Simulink models, an automated technique to support localization of faults in Simulink models is crucial.

Fault localization in source code is an active research area that focuses on automating various code debugging activities [5, 10, 13, 14, 29, 30, 32, 42, 43, 11, 39]. A well-known approach in this area is *statistical debugging* [32, 19, 20, 21, 2, 39]. Statistical debugging is a light-weight approach to fault localization and has been extensively studied for code (e.g., C programs [32, 19, 2]). 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 ranking to identify faults in their code. These fault localization techniques, however, have never been studied for Simulink models.

In this paper, we propose *SimFL*, our approach to localize faults in Simulink models based on statistical debugging. Our approach, while being based on statistical debugging, takes into account the characteristics of Simulink models. 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 the existing work.

Simulink models, being visual, data-flow based and hierarchical, have multiple observable outputs at different hierarchy levels, each of which can be tested and evaluated independently. For Simulink models, engineers not only identify whether a test case passes or fails, but they also routinely and explicitly determine which specific outputs are correct and which ones are incorrect for each given test case. 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 model blocks using statistical ranking formulas. In this work, we consider three well-known statistical formulas used for fault localization for source code (i.e., *Tarantula* [19], *Ochiai* [2], and *Naish2* [28]), and study their accuracy in localizing faults in Simulink models. Nevertheless, our approach is not specific to any particular statistical formula and other ones could also be used.

Our approach relies on accounting for as many outputs as possible to increase the number of spectra used to compute rankings. Such expansion, however, should be driven by needs to avoid unreasonable overhead resulting from checking more test oracles on test case executions. In our work, we propose *iSimFL* where we apply *SimFL* iteratively starting with a small test oracle size to obtain some initial ranking results. We provide a heuristic to guide engineers based on the quality of the rankings obtained at each iteration to decide whether they need to extend the oracle or not. We then apply *SimFL* iteratively until the ranking results are satisfactory and do not require any further oracle expansion. In this paper, we make the following contributions:

(1) We propose *SimFL*, a new combination of statistical debugging and dynamic slicing to localize faults in Simulink models. Even though dynamic slicing has been used together with statistical debugging [3, 15, 24, 27], our work is the first to define the notion of spectrum per test case and per output as opposed to the existing work where there is a one-to-one correspondence between test suites and sets of spectra.

(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. Our experiments show that: **(i)** On average, using *SimFL* with *Tarantula*, engineers need to inspect at least 2.1% and at most 8.9% of Simulink models to find faults. Further, we found that the accuracy of *Tarantula*, *Ochiai* and *Naish2* in localizing faults in Simulink models are considerably close. **(ii)** We show that increasing the size of test suites does not make any significant improvement in *SimFL*'s accuracy. **(iii)** We show that in most (but not all) cases, extending the test oracle to include outputs at lower hierarchy levels significantly improves the fault localization capability of *SimFL*. Specifically, *iSimFL* improves *SimFL*'s accuracy by requiring engineers to inspect, on average, at least 1.3% and at most 4.4% of the model blocks while extending test oracles with only five outputs on average. **(iv)** We investigate the predictability of *iSimFL*'s performance with respect to changes made in to its input heuristic parameters. We, further, provide suggestions, based on the results of our experiments, on how to set these parameters to obtain optimal results.

The paper is organized as follows: In Section 2, we provide background on Simulink models and motivate our approach. In Section 3, we present *SimFL*, our approach to fault localization in Simulink models. In Section 4, we present an iterative fault localization approach, namely *iSimFL* that can further improve the accuracy of *SimFL* in localizing faults. In Section 5, we evaluate our work and discuss our experiment results. We then compare our work with the related work in Section 6. We conclude the paper and discuss our future work in Section 7.

2 Background and Motivation

In this section, we briefly introduce a Simulink model example, describe how Simulink models are tested/simulated in practice, and provide a brief overview of our approach.

Simulink model example. Figure 1 presents a small sanitized snippet of a real-world Simulink model from Delphi. The specific fragment in Figure 1 calculates temperature and pressure of the compressed air. The model is essentially composed of numerical and combinatorial operations as well as some constant blocks, e.g., `Pmax`. The model has five inputs, e.g., the position of the `Clutch`, and two outputs: the output pressure `pOut` and the output temperature `TOut`. The model inputs are specified by input ports (dashed rounded boxes), and the model outputs are shown using output ports (grey rounded boxes). Further, Simulink models are hierarchical and allow the encapsulation of blocks into subsystems. For the model in Figure 1, there are two subsystems: `Subsystem1` and `Subsystem2`. Each subsystem has its own input and output ports. Each input and output port of a subsystem is connected to an atomic block within the subsystem.

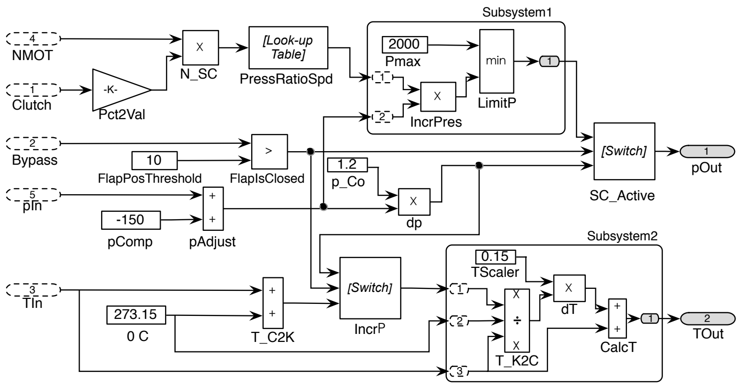


Figure 1: A snippet of a real-world Simulink model.

Input data. Engineers simulate (execute) the model in Figure 1 by providing five 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(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.

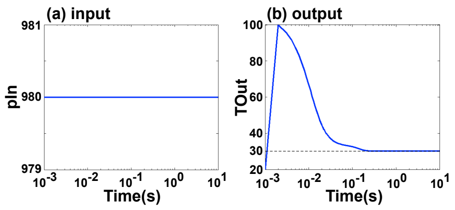


Figure 2: Example of an input (a) and an output (b) signal.

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(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(b)).

Overview of Our Approach. Simulink models consist of blocks and lines. Blocks represent individual functions, whereas lines represent data and control flow relations between blocks. Given a Simulink model with multiple outputs (e.g., Figure 1), for a single test case execution, it often happens that some outputs reveal failures, while others are correct. If any individual output value deviates from its oracle, the engineers typically follow the links connected to that output in a backward manner to identify faulty block(s). This helps engineers focus only on blocks that can reach the erroneous outputs via data and control dependency links. For example, in Figure 1, if `pOut` shows an error for a test case, we know that the failure at `pOut` cannot be due to a fault at `T_C2K` because `T_C2K` is not in static backward slice of `pOut`.

Inspired by how Simulink models are debugged in practice, in our work, we define the notion of spectrum per test case and per output. Specifically, we propose in Section 3.2 a dynamic slicing approach to identify, for a given Simulink model, *test execution slices* of that model for each test case and each output. Each test execution slice (spectrum) represents the set of blocks that are executed by a single test case to produce a specific output. As a result, the number of spectra we obtain is larger than the size of the test suite, and hence, we have more observation points to perform fault localization. We propose (in Section 4) to apply our fault localization approach iteratively starting with a small set of outputs. Further, we provide a heuristic which, based on the ranking results from the previous iteration, determines whether extending test oracle is needed or not. If so, engineers extend the output set typically by including subsystem outputs, and perform another fault localization iteration.

Our iterative fault localization approach requires engineers to develop test oracles for several model outputs as well as subsystem outputs. We note that for Simulink models from the automotive domain, test oracles are typically developed not just for final model outputs (e.g., `pOut` and `TOut` in Figure 1), but also for subsystem outputs (e.g., the outputs of `Subsystem1` and `Subsystem2` in Figure 1). This is because in this domain Simulink models often capture physical devices (e.g., a supercharger). Specifically, a Simulink subsystem often conceptually relates to a real stand-alone entity, and its outputs are meaningful and can be evaluated independently from the final model outputs. Finally, in our approach, we provide a heuristic to guide engineers on when test oracle expansion is unlikely to be worthwhile, thus avoiding unnecessary overhead.

3 Fault Localization for Simulink

We present *SimFL*, our fault localization approach for Simulink models. Figure 3 shows an overview of *SimFL*. The inputs to our approach is a (faulty) Simulink model (M), a test suite ($TS = \{tc_0, \ldots, tc_n\}$), 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 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 $tc \in TS$,

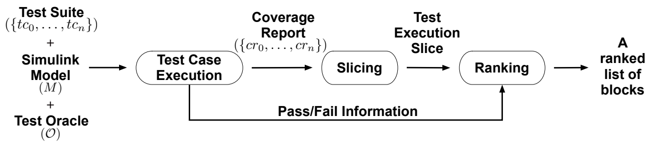


Figure 3: Overview of our fault localization approach for Simulink (*SimFL*).

the test oracle $\mathcal{O}$ determines whether each output $o \in O$ passes or fails tc.

The output of the approach in Figure 3 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 3.1 to 3.3, respectively.

3.1 Test Case Execution

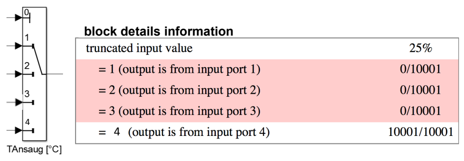
This step takes as input a test suite TS, a test oracle $\mathcal{O}$, and a (faulty) Simulink model M. In this step, we execute M for each test case in TS to generate the following information: (1) The `PASS`/`FAIL` information corresponding to each output o of M and each test case in TS, and (2) A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

In Section 2, 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. 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 4 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 4 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 4, we conclude that among inputs 0–4, only control input 0 and data input 4 were covered during simulation.

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

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



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 4: A coverage report snippet generated by Simulink

which specific output. We use slicing (Section 3.2) to determine which blocks were covered for which output.

3.2 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 $\{cr_0, \ldots, cr_n\}$ 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 [31]. 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 1). Finally, we note that Simulink models may contain feedback loops enabling to use the output of a block for a subsequent calculation [31]. 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 5 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 5 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.

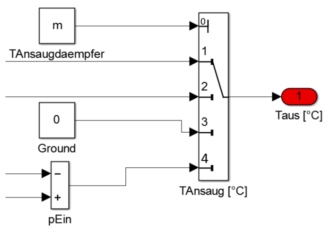


Figure 5: A (partial) static slice for a MultiPortSwitch block.

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. That is, $slice_{o,l} = \{b \in B \mid b \text{ is covered by } tc_l \text{ to generate } o\}$, where B is the set of atomic blocks in M. We compute a test execution slice $slice_{o,l}$ by traversing the blocks in the static backward slice of 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. That is, 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 3.1, the coverage report in Figure 4 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 5, 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.

Table 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 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 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.

3.3 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* [19], *Ochiai* [2], and *Naish2* [28]. *Tarantula* and *Ochiai* have been

Table 1: Test execution slices and suspiciousness scores of model blocks using *Tarantula* for the example model of Figure 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				

the subject of many experiments, and are supported by more substantial empirical evidence than other formulas [18, 33, 16, 17, 6, 40, 28, 23, 22]. Recently, *Naish2* has been shown to be a theoretically optimal formula for fault localization [28, 40], and further, it has also shown to perform well empirically [23]. 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 $\mathit{passed}(s)$ and $\mathit{failed}(s)$ 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*, *Naish2*, denoted by $\mathit{Score}^{Ta}(s)$, $\mathit{Score}^{Oc}(s)$, and $\mathit{Score}^{N2}(s)$, respectively, are calculated as:

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

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

$$\mathit{Score}^{N2}(s) = \mathit{failed}(s) - \frac{\mathit{passed}(s)}{\mathit{totalpassed} + 1}$$

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, $\mathit{totalpassed}_o$, $\mathit{totalfailed}_o$, passed_o, and failed_o for every output $o \in O$. Based on the Test Case Execution step, we already have the total number of passed ($\mathit{totalpassed}_o$) and the total number of failed ($\mathit{totalfailed}_o$)

test execution slices per each output $o \in O$. We define the functions $passed_o$, $failed_o$ for every $o \in O$ as follows:

$passed_o(b) = |\{slice_{o,l} \mid b \in slice_{o,l} \wedge slice_{o,l} \text{ is passing}\}|$

$failed_o(b) = |\{slice_{o,l} \mid b \in slice_{o,l} \wedge slice_{o,l} \text{ is failing}\}|$

That is, $passed_o(b)$ and $failed_o(b)$ represent the number of passing and failing test execution slices $slice_{o,l}$ that include b, respectively. Note that 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. For each output $o \in O$, we define the suspiciousness score of block b for *Tarantula*, $Score_o^{Ta}$, for *Ochiai*, $Score_o^{Oc}$, and for *Naish2*, $Score_o^{N2}$, as follows:

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

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

$$Score_o^{N2}(b) = failed_o(b) - \frac{passed_o(b)}{totalpassed_o + 1}$$

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 5, 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 5.

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

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 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 1. Note that undefined scores are shown as *NaN* cells and are not used for mean score computation. Table 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.

4 Iterative Fault Localization

In this section, we describe how the approach in Figure 3 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 6 shows our iterative fault localization algorithm referred to as *iSimFL*. Similar to *SimFL* (Figure 3), *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.

As discussed in Section 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 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 6 implement our heuristic.

```
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 $|\bigcup_{0 \le i \le k} g_i| \ge N$ and $|\bigcup_{0 \le i \le k-1} g_i| < N$
6.    Let $g^*$ be the largest group in $L'$
7.    if $(|g^*| < g)$ do
8.      break;
9.    $itr++$
10. while $(itr \le h)$
11. return $L$
```

Figure 6: Iterative fault localization with *iSimFL*

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).

5 Empirical Evaluation

In this section, we present our research questions, describe our industrial subjects, and experimental setup, followed by the analysis of the results.

5.1 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*? We start by investigating 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 *Naish2*. 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. Here, we focus 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. We investigate 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.

5.2 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 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 1, the `Subsystem2` output matches `TOut`).

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

Table 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

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 [26], 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 paper. 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 [25]. 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.

5.3 Experiment Settings

In addition to a Simulink model, which is discussed in Section 5.2, *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 paper, we generated test suites using Adaptive Random Testing [8]. 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 paper. 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(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 four 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 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) using *Tarantula*, *Ochiai*, and *Naish2* 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. Based on the results of *EXP1*, we select one statistical formula to be used by *SimFL* in *EXP2* to *EXP5.* For **RQ2**, we perform experiment *EXP2* where we apply *SimFL* 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* to our 95 faulty models with a test suite size of 200 and with test oracles $\mathcal{O}_1$ and $\mathcal{O}_2$. For **RQ4**, we perform experiment *EXP4* where we apply *iSimFL* (Figure 6) 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 [30, 32, 21, 11, 18, 23]: 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.

5.4 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 5.3. 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, 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 *Naish2*. 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 *Naish2*, engineers need to inspect, on average, at least 16 and at most 55 (i.e., 2.4% - 8%).

Table 3: Average percentage and absolute number of blocks inspected using *SimFL* with $\mathcal{O}_0$ for *Tarantula*, *Ochiai*, and *Naish2*.

Model name	min.#(%) - max.#(%) for *SimFL* with $\mathcal{O}_0$		
	Tarantula	*Ochiai*	*Naish2*
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%)	24(2.9%) - 83(10.2%)
MG	4 (1.4%) - 18(6%)	5(1.6%) - 18(6%)	8(2.7%) - 22(7.3%)
All models	14 (2.1%) - 63 (8.9%)	23 (3.1%) - 62 (8.8%)	16 (2.4%) - 55 (8%)

Proportion of faults localized. In Figures 7, 8, and 9, we present the proportion of faults localized when engineers inspect a fixed number of the most suspicious blocks in the rank list generated by *SimFL* for *Tarantula*, *Ochiai*, and *Naish2*, 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.

Using *SimFL* with *Tarantula* (see Figure 7), engineers who inspect the top 10% of most suspicious blocks (i.e., 65 blocks of *MS*, 82 blocks of *MC* and 30 blocks of *MG*), can locate, for *MS*, at most 95% and at least 78% of the faults; for *MC*, at most 85% and at least 33% of the faults; and for *MG*, at most 100% and at least 93% of the faults. Using *SimFL* with *Ochiai* (see Figure 8), engineers who inspect the top 10% of most suspicious blocks, can locate, for *MS*, at most 92.5% and at least 77.5% of the faults; for *MC*, at most 80% and at least 32.5% of the faults; and for *MG*, at most 100% and at least 86.6% of the faults. Using *SimFL* with *Naish2* (see Figure 9), engineers who inspect the top 10% of most suspicious blocks, can locate, for *MS*, at most 97.5% and at least 82.5% of the faults; for *MC*, at most 87.5% and at least 37.5% of the faults; and for *MG*, at most 100% and at least 73.3% of the faults. Finally, when engineers inspect the top 10% of most suspicious blocks, engineer can localize at least 58 and at most 87 faults out of 95 faults when using *SimFL* with *Tarantula*, at least 57 and at most 84 faults out of 95 faults when using *SimFL* with *Ochiai*, and at least 59 and at most 89 faults out of 95 faults, when using *SimFL* with *Naish2*.

Note that, for all of the three formulas (i.e., *Tarantula*, *Ochiai*, and *Naish2*), 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, on average, according to [32, 21, 11, 23], developers need to inspect at most around 20% of their code (i.e., program blocks) to localize faults, while *SimFL*, on average, requires at most around 8% (i.e., 8.9% for *SimFL* with *Tarantula* 8.8% for *SimFL* with *Ochiai*, and 8% for *SimFL* with *Naish2*) 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% [32, 21, 18, 23]. 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 58/95 for *SimFL* with *Naish2*).

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* Naish2. The above results show that the accuracy of ranking results obtained by these three formulas are considerably close. Based on Table 3, the percentage and absolute numbers of blocks inspected using *SimFL* with $\mathcal{O}_0$ for *Tarantula*, *Ochiai*, and *Naish2* are considerably close. The average maximum percentages of blocks inspected corresponding to the three formulas range from 8% to 9%. Figure 10 shows the comparison of the minimum proportions of faults localized for all 95 faulty versions when using *SimFL* with *Tarantula*, *Ochiai*, and *Naish2*. Based on Figure 10, the min-

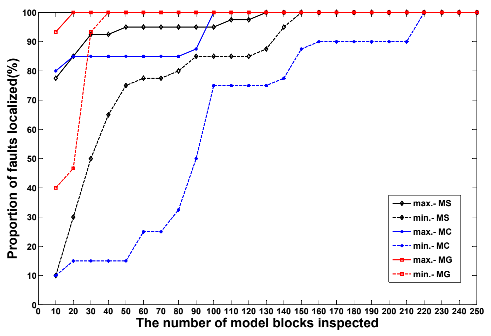


Figure 7: Proportion of faults localized for *SimFL* with $\mathcal{O}_0$ and *Tarantula* as the statistical formula.

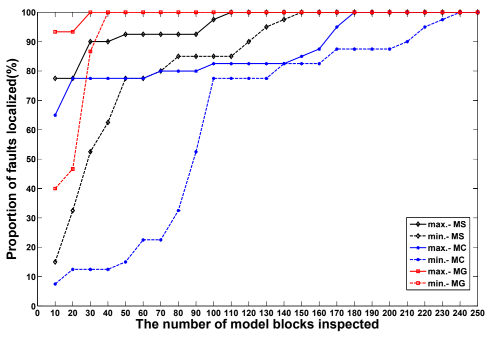


Figure 8: Proportion of faults localized for *SimFL* with $\mathcal{O}_0$ and *Ochiai* as the statistical formula.

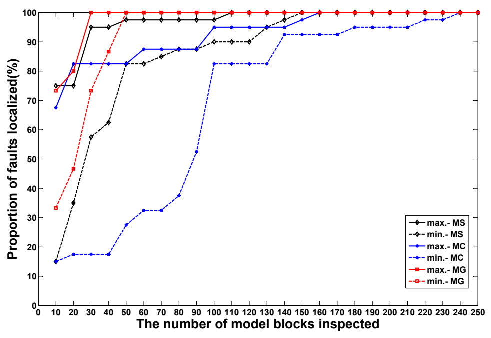


Figure 9: Proportion of faults localized for *SimFL* with $\mathcal{O}_0$ and *Naish2* as the statistical formula.

imum proportions of faults localized when using the three formulas are close, in particular when engineers inspect the top 40 blocks. Further, when engineers inspect more than 40 blocks, the variations in the minimum proportions of faults localized across the three formulas are less than 9%, and hence, not substantial.

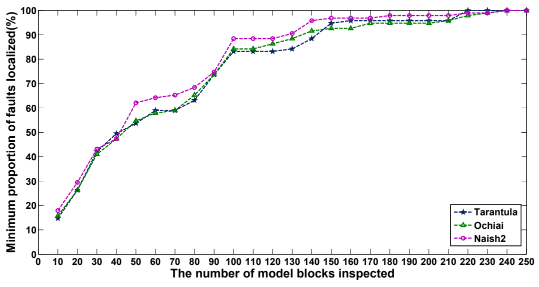


Figure 10: Comparison of minimum proportion of faults localized for *SimFL* ($\mathcal{O}_0$) with *Tarantula*, *Ochiai*, *Naish2*.

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 *Naish2* in localizing faults in Simulink models are not substantially different.

Since the accuracies of *Tarantula*, *Ochiai*, and *Naish2* in localizing faults in Simulink models are close, we answer the following research questions based on the results *SimFL* with Tarantula. We believe that the answers to RQ2 to RQ5 would remain the same when we use *SimFL* with the other two formulas.

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 test suite sizes of 200, 300, . . . , 1000. For all the 95 faulty models, changes in the scores and the rankings of the faulty blocks are negligible as we apply *SimFL* with different test suite sizes. 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 investigate the number of *Coincidentally Correct Test cases (CCT)* [38]. 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. Based on the *Tarantula* formula and given that our faulty models include a single faulty block, the scores of faulty blocks depend on the proportion of CCTs over the total number of passing test execution slices [38]. In our experiment, we observed that as we add more test cases the proportion of CCT over all passing test execution slices remains almost constant (i.e., changes in this proportion are less than 1%). Therefore, the ranks of faulty blocks remain nearly constant. Note that, we most likely obtain similar results with *Ochiai* and *Naish2* because, like *Tarantula*, the scores generated by these two formulas also depend on the proportion of CCTs over the total number of passing test execution slices.

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 11(a) to 11(f), we show the maximum percentages of blocks required to be inspected for 58 out of 95 faulty models when *SimFL* 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 11(a) and 11(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 11(c) to 11(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 11(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 11(b) to 11(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 11(b), 11(c), and 11(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 11(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 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 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 11(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

Table 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. ~ max.)		
		Test oracle $\mathcal{O}_0$	Test oracle $\mathcal{O}_1$	Test oracle $\mathcal{O}_2$
MS	200	6 ~ 1009	34 ~ 1796	47 ~ 2461
MC	200	8 ~ 1010	8 ~ 1390	8 ~ 1390
MG	200	34 ~ 249	92 ~ 467	102 ~ 467

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 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, 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 $\mathcal{O}_0$.

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.

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 11), *iSimFL* only performed one iteration before it terminates. That is, the loop in Figure 6 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 11(g) to 11(l).

For 16 faulty models as shown in Figure 11(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 test oracle to $\mathcal{O}_2$.

For four faulty models as shown in Figure 11(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

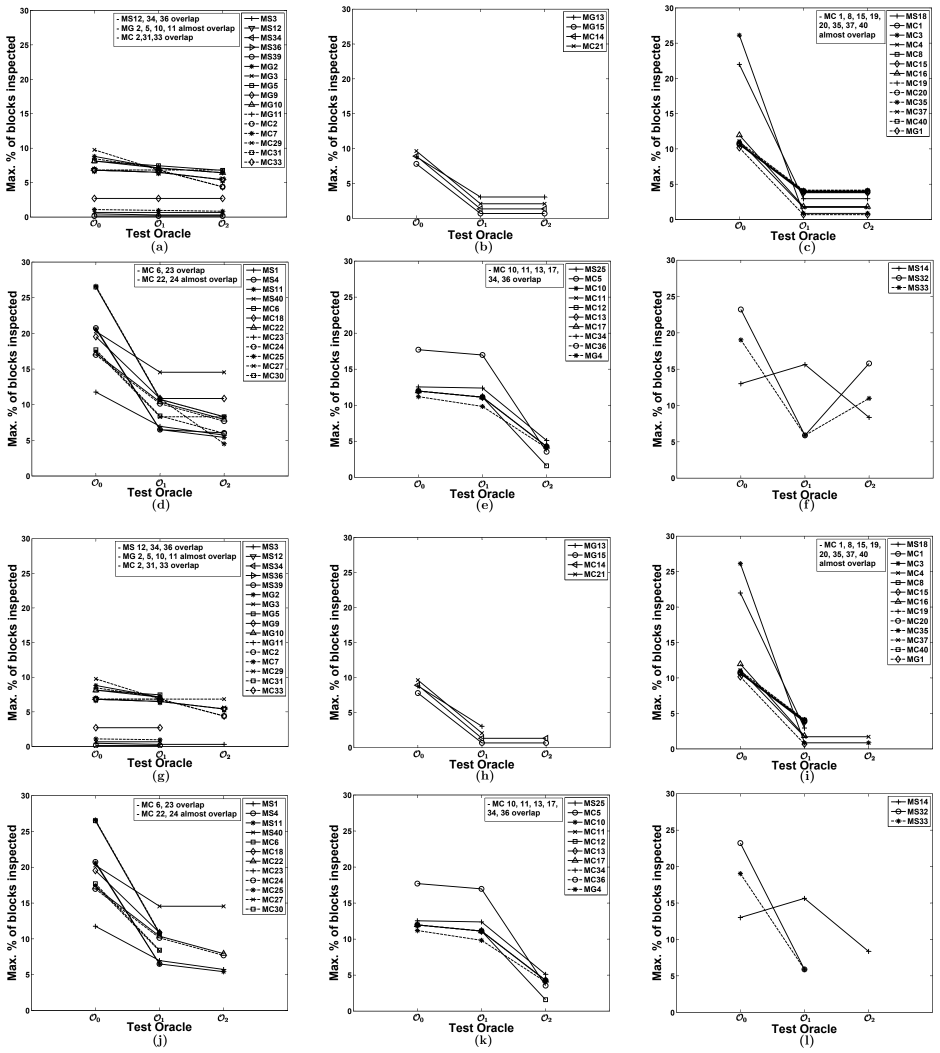


Figure 11: Maximum percentage of blocks that need to be inspected to find faults for *SimFL* with test oracles $\mathcal{O}_0$, $\mathcal{O}_1$, and $\mathcal{O}_2$ and for *iSimFL*: (a) *SimFL*'s accuracy improves slightly or remains the same as we extend the oracle, (b-e) *SimFL*'s accuracy improves 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.

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 11(i) to 11(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 11(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 11(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 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 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 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,

Table 5: Average of minimum and maximum numbers of blocks inspected and test oracle sizes when using *SimFL* with $\mathcal{O}_0$, *SimFL* with $\mathcal{O}_2$, and *iSimFL*.

Model name	*SimFL* with $\mathcal{O}_0$	*SimFL* with $\mathcal{O}_2$	*iSimFL*
	min. #(%) - max. #(%) ($\|\mathcal{O}_0\|$)	min. #(%) - max. #(%) ($\|\mathcal{O}_2\|$)	min. #(%) - max. #(%) (Avg.$\|\mathcal{O}\|$)
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)

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 5.3. 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 12 and 13 show the results of these experiments: In Figure 12, 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 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 12(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 12(b)). But *iSimFL* obtains these rankings with an oracle that contains on average seven less outputs compared to $\mathcal{O}_2$ (see Figure 12(a)). In Figure 13, 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 12, Figure 13(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 13(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 12, 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 $\mathcal{O}_2$ 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

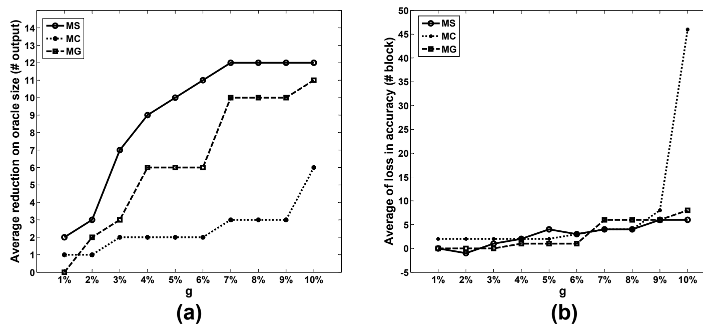


Figure 12: The impact of varying the value of g on the average reduction of oracle size (a) and the average loss in fault localization accuracy (b).

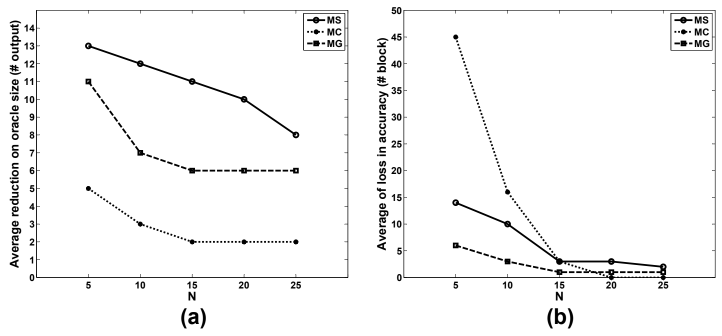


Figure 13: The impact of varying the value of N on the average reduction of oracle size (a) and the average loss in fault localization accuracy (b).

extend test oracles less often (line 7 in Figure 6). 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 12(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 13) 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 13(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 13(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 12, 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 13, 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 11, 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. Below 4%, our results suggest that the reduction in oracle size will be limited. As for N, engineers are limited by the time they can dedicate to inspecting blocks but our results suggest that N should be at least 15, irrespective of the size of the model.

5.5 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 [32, 19, 20, 21, 2, 18, 7, 22, 41, 33, 9, 4, 30, 10, 11]. 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.

6 Related Work

In this section we present the related work to our fault localization approach. First, we discuss fault localization techniques applied to source code that are closely related to our work. Next, we present the existing work on the analysis of Simulink models.

6.1 Software Fault Localization

Many fault localization techniques have been proposed to localize faults in programs[5, 10, 13, 14, 29, 30, 32, 42, 43, 11, 39, 19, 20, 21, 1, 2, 18, 7, 22, 41, 33, 9, 4]. Statistical debugging is one family of fault localization approaches that has been extensively studied to localize faults in programs [32, 19, 20, 21, 1, 2, 18, 7, 22, 41, 33, 9, 4]. 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 [32, 19, 18, 41], program blocks [2, 1, 22, 32], predicates [20, 21], combination of spectra [33], program path [9]. A number of statistical formulas to measure suspiciousness of program elements have also been proposed e.g., *Tarantula* [19], *Ochiai* [2, 1], formulas from data mining [22], Naish [28], formulas generated using genetic programming [41], SOBER [21], CBI [20]. In this work, we analyze sequences of (atomic) blocks in Simulink and use existing statistical formulas (i.e., Tarantula, Ochiai, and Naish) to measure the suspiciousness of Simulink (atomic) blocks to be faulty.

The above techniques [32, 19, 20, 21, 1, 2, 18, 7, 22, 41, 33] localize faults by performing statistical debugging technique only once. Other debugging techniques [44, 4, 9] iteratively apply a statistical debugging technique until developers find the root cause of failures. The techniques proposed in [9, 4] 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. [9] 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. Nainar and Liblit [4] 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. [44] 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 [9, 4, 44] 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. [12] 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 [3, 15, 24, 27]. Mao et al [24] use static backward slicing to prune the executed statements that do not impact any output, and rank the statements using statistical debugging. In contrast, Hofer et al. [15] first obtain suspiciousness scores for statements, and compute a minimum set of faulty statements using dynamic slicing. Statistical ranking results can be refined by applying a model-based approach to remove program elements that do not relate to failures and then computing a minimum set of faulty elements[3, 27]. The above existing approaches produce a single spectrum for each test case. In contrast, we redefine the notion of spectrum as a set of Simulink blocks that are executed by each test case to generate a specific output. This enables engineers to obtain finer grained spectra information about individual outputs even when available test suites are small and cannot be extended arbitrarily due to practical limits of embedded system development. We compute suspiciousness scores for each block and each output, and take the average of suspiciousness scores of each block over all outputs to obtain final scores used for ranking.

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 [30]. 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 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.

6.2 Analysis of Simulink Models

In our work, we relied on model simulations to identify control dependencies between Simulink blocks. Reicherdt and Glesner [31] 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 of [31, 36] 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 [34] 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 5.2). Further, in [34], the author 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 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 *Naish2* 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.

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.

Acknowledgement

We thank Mehrdad Sabetzadeh and anonymous reviewers for their useful comments on an earlier draft of this paper. This work is supported by the Fonds National de la Recherche, Luxembourg (FNR/P10/03 and FNR8003491), and Delphi Automotive Systems, Luxembourg.

References

[1] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[2] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 89–98. IEEE, 2007.

[3] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 88–99. IEEE, 2009.

[4] Piramanayagam Arumuga Nainar and Ben Liblit. Adaptive bug isolation. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 255–264. ACM, 2010.

[5] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 97–105, 2003.

[6] Benoit Baudry, Franck Fleurey, and Yves Le Traon. Improving test suites for efficient fault localization. In *Proceedings of the 28th international conference on Software engineering*, pages 82–91. ACM, 2006.

[7] Mike Y Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings. International Conference on Dependable Systems and Networks, 2002. DSN 2002.*, pages 595–604. IEEE, 2002.

[8] Tsong Yueh Chen, Hing Leung, and IK Mak. Adaptive random testing. In *Advances in Computer Science-ASIAN 2004. Higher-Level Decision Making*, pages 320–329. Springer, 2005.

[9] Trishul M Chilimbi, Ben Liblit, Krishna Mehra, Aditya V Nori, and Kapil Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 34–44. IEEE, 2009.

[10] Holger Cleve and Andreas Zeller. Finding failure causes through automated testing. In *Proceedings of the Fourth International Workshop on Automated Debugging*, AADEBUG 00, 2000.

[11] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 342–351. ACM, 2005.

[12] Liang Gong, Daniel Lo, Lingxiao Jiang, and Hongyu Zhang. Interactive fault localization leveraging simple user feedback. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 67–76. IEEE, 2012.

[13] Alex Groce, Daniel Kroening, and Flavio Lerda. Understanding counterexamples with explain. In *Computer Aided Verification*, pages 453–456. Springer, 2004.

[14] Ralf Hildebrandt and Andreas Zeller. Simplifying failure-inducing input. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, pages 135–145, 2000.

[15] Birgit Hofer and Franz Wotawa. Spectrum enhanced dynamic slicing for better fault localization. In *ECAI*, pages 420–425, 2012.

[16] Hwa-You Hsu, James A Jones, and Alessandro Orso. Rapid: Identifying bug signatures to support debugging activities. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 439–442. IEEE Computer Society, 2008.

[17] James A Jones, James F Bowring, and Mary Jean Harrold. Debugging in parallel. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 16–26. ACM, 2007.

[18] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 273–282, 2005.

[19] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, 2002.

[20] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. Scalable statistical bug isolation. *ACM SIGPLAN Notices*, 40(6):15–26, 2005.

[21] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P Midkiff. Sober: statistical model-based bug localization. *ACM SIGSOFT Software Engineering Notes*, 30(5):286–295, 2005.

[22] David Lo, Lingxiao Jiang, Aditya Budi, et al. Comprehensive evaluation of association measures for fault localization. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[23] Lucia, David Lo, and Xin Xia. Fusion fault localizers. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 127–138, 2014.

[24] Xiaoguang Mao, Yan Lei, Ziying Dai, Yuhua Qi, and Chengsong Wang. Slice-based statistical fault localization. *Journal of Systems and Software*, 89:51–62, 2014.

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

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

[27] Wolfgang Mayer, Rui Abreu, Markus Stumptner, AJ van Gemund, et al. Prioritising model-based debugging diagnostic reports. In *Proceedings of the International Workshop on Principles of Diagnosis (DX)*, pages 127–134, 2009.

[28] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)*, 20(3):11, 2011.

[29] Alessandro Orso, James A. Jones, Mary Jean Harrold, and John T. Stasko. Gammatella: Visualization of program-execution data for deployed software. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 699–700, 2004.

[30] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 20th International Symposium on Software Testing and Analysis*, ISSTA '11, pages 199–209, 2011.

[31] R. Reicherdt and S. Glesner. Slicing matlab simulink models. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 551–561, 2012.

[32] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, ASE '03, pages 30–39, 2003.

[33] Raul Santelices, James A Jones, Yanbing Yu, and Mary Jean Harrold. Lightweight fault-localization using multiple coverage types. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 56–66. IEEE, 2009.

[34] Johanna Schneider. Tracking down root causes of defects in simulink models. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 599–604, 2014.

[35] P Skruch, M Panek, and B Kowalczyk. Model-based testing in embedded automotive systems. *Model-Based Testing for Embedded Systems*, pages 293–308, 2011.

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

[37] Andreas Thums and Jochen Quante. Reengineering embedded automotive software. In *Proceedings of the 28th IEEE International Conference on Software Maintenance*, ICSM '12, pages 493–502, 2012.

[38] Xinming Wang, Shing-Chi Cheung, Wing Kwong Chan, and Zhenyu Zhang. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 45–55. IEEE Computer Society, 2009.

[39] Eric Wong, Tingting Wei, Yu Qi, and Lei Zhao. A crosstab-based statistical method for effective fault localization. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, ICST '08, pages 42–51, 2008.

[40] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4):31, 2013.

[41] Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, Shin Yoo, and Mark Harman. Provably optimal and human-competitive results in sbse for spectrum based fault localisation. In *Search Based Software Engineering*, pages 224–238. Springer, 2013.

[42] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Pruning dynamic slices with confidence. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 169–180, 2006.

[43] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Precise dynamic slicing algorithms. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 319–329, 2003.

[44] Zhiqiang Zuo, Siau-Cheng Khoo, and Chengnian Sun. Efficient statistical debugging via hierarchical instrumentation. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 457–460. ACM, 2014.