Localizing Multiple Faults in Simulink Models

Bing Liu,*, Lucia*, Shiva Nejati*, Lionel Briand*, Thomas Bruckmann†
* SnT Centre, University of Luxembourg, Luxembourg
Email: {bing.liu, lucia.lucia, shiva.nejati, lionel.briand}@uni.lu
†Delphi Automotive Systems, Luxembourg
Email: thomas.bruckmann@delphi.com

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

***Keywords*—Fault localization, statistical debugging, machine learning, decision trees, Simulink models.**

I. Introduction

Simulation or *design-time testing of system models* is becoming an indispensable activity when developing complex systems [1]–[4]. Among the languages that are typically used for simulation, Simulink [5] is a notable example. Simulink attempts to combine the benefits of modeling (abstraction) and programming (executability) to facilitate early system specification and testing. Despite wide-spread use of Simulink in embedded system development, there is no automated support for debugging Simulink models.

Debugging is a cumbersome and time-consuming task. There is a wide range of techniques in the literature for debugging and fault localization in source code [6]–[18]. However, none of these techniques have been previously applied to Simulink models. In our recent work [19], we proposed an approach based on *statistical debugging* to automatically localize *individual* faults in Simulink models. Statistical debugging is a light-weight and well-studied approach to fault localization in code (e.g., C programs [12], [15], [18]). 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 failing or passing test cases, are used to derive a statistical fault *ranking*, specifying an ordered list of program elements (e.g., statements) likely to be faulty. Engineers consider such ranking to identify faults in their code.

Statistical debugging often fails to properly deal with multiple faults because it implicitly assumes that all failures are caused by the same fault(s). However, in the presence of multiple faults, different failures might be due to different faults and faults might mask one another. A number of techniques aim to improve statistical debugging to handle multiple faults [16], [20]–[28]. Most of these techniques [20]–[25] cluster failures such that the failures that are grouped together are likely to have been caused by the same fault(s), and hence, can be used specifically to localize those faults. For each cluster, a ranked list of most suspicious elements is generated. Engineers inspect *all* these ranked lists to identify faults [20]–[22]. Alternatively, some techniques [23]–[25] generate a single ranking based on the clustering results to be inspected by engineers.

In this paper, we propose a statistical debugging approach that clusters failures to help identify multiple faults in Simulink models. Simulink models have multiple observable outputs, each of which can be tested and evaluated independently. Hence, a failing test case may result in several failures. Specifically, we associate a failure with a *failing execution slice* denoted by a pair (tc, o) where tc is a failing test case and o is a model output at which tc fails. This is in contrast to the existing approaches [20]–[25], [29] that associate a failure with a failing test case.

Unlike the existing approaches [21], [22], [24], [25] that use unsupervised learning for failure clustering, we use a supervised learning technique, namely *decision trees* [30]. Using decision trees, we group together failing execution slices that satisfy similar logical conditions on model blocks and on model inputs. Our decision trees extract the most relevant information from failing and passing test executions and, in contrast to unsupervised learning techniques, do not require similarity measures to be defined a priori. Further, the approaches based on unsupervised learning should specify a cut threshold on similarity measures to form clusters. Our work does not require such thresholds. Instead, we define a termination criterion based on the size and the degree of homogeneity of clusters. Finally, the input to our decision tree-based approach contains both categorical (sets of blocks covered by execution slices) and numerical variables (model inputs), while existing approaches focus on either of these items and have never considered both together.

For each cluster generated by our decision trees, we use the statistical formula, *Tarantula* [15], [31], to generate a ranked list of most suspicious Simulink blocks. Note that our comparison [19] of alternative statistical formulas applied to Simulink models revealed no significant difference among these formulas and *Tarantula*. Some existing approaches [20]–[22], [29] for localizing multiple faults require engineers to inspect all ranked lists obtained from all the clusters to identify the top-most ranked fault in each list. This approach to debugging, however, may miss the faults that are masked by others, and hence, are ranked lower than others in all the ranked lists. Further, the top-most ranked faults in several ranked lists may overlap. As a result, requiring engineers to inspect all the ranked lists may lead to identifying duplicate faults, hence wasting their effort.

In our work, instead, we adopt a debugging process referred to as *one-at-a-time* debugging [32]. That is, our fault localization approach is applied iteratively where at each iteration, engineers identify and resolve one fault. Engineers then re-test the model to ensure that particular fault is fixed, and to focus on localizing other faults including those that might have been masked or were not observable in the previous round of testing. Note that the one-at-a-time debugging process matches the practice in the development of Simulink models, and impacts how we evaluate and compare fault localization techniques. To be able to utilize our technique in the context of one-at-a-time debugging, we introduce two selection criteria to choose a ranked list that is likely to yield the best fault localization results among other ranked lists. In particular, our criteria aim to select the most coherent clusters, i.e., those with the most similar failing execution slices. Engineers inspect the selected ranked list to localize faults, and re-test the model and apply our approach until no failures are observed.

In summary, the contributions of this paper are as follows:

(1) We propose a new iterative technique to localize multiple faults in Simulink models using a supervised learning technique (decision trees).

(2) We evaluate our approach on 240 multi-fault Simulink models obtained from three different industrial subjects. We have further adapted and implemented two baseline techniques for Simulink models: A traditional statistical debugging approach without clustering, and a state-of-the-art clustering-based statistical debugging that uses unsupervised learning [22]. We compared our decision tree-based approach with the two baseline techniques and our results show that: (I) Our approach is able to significantly reduce the number of blocks that are required to be inspected to localize all faults compared with the two baselines. (II) With our approach, there is less performance degradation than in the baselines when increasing the number of faults in the underlying models. That is, our approach is more robust than the baselines when applied to models containing larger numbers of faults.

II. Background

In this section, we briefly describe Simulink models and statistical debugging for Simulink models.

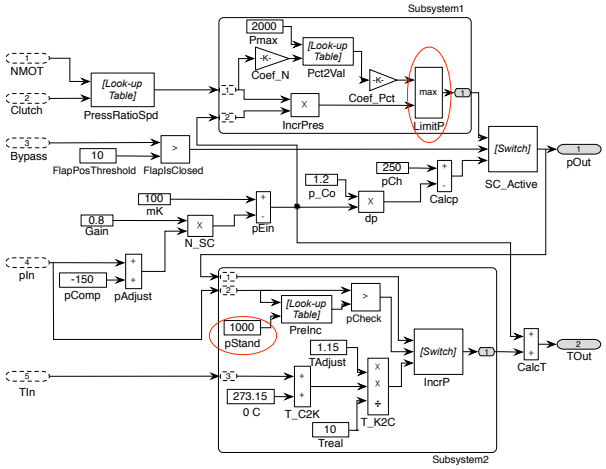


Fig. 1: A Simulink model example with faulty blocks `LimitP` and `pStand`.

Simulink models. Figure 1 shows an example of a Simulink model. Simulink models consist of blocks and lines. Blocks may perform individual operations such as numerical and combinatoric operations or they may represent constant values e.g., the `pStand` block. Simulink blocks are connected via lines that indicate data flow connections. The model in Figure 1 has five inputs e.g., the position of the `Clutch`, and two outputs: the output temperature `TOut` and pressure `pOut`. The inputs are specified by input ports (dashed rounded boxes), and the outputs are shown using output ports (grey rounded boxes).

Given a Simulink model with multiple outputs (e.g., Figure 1), any individual test case execution may show failures in some outputs, while other outputs might be correct for that particular test case. For example, in Figure 1, there are two faults in the model: The value of the constant block `pStand` is set to 1000 instead of 980, and the `LimitP` block performs a max operation instead of a min operation. Given any test case, failures may be observed at either or both outputs of the model in Figure 1. Once a failure is observed at a particular output, engineers require some technique to help with debugging.

Statistical debugging for Simulink. In our previous work [19], we have shown how statistical debugging can be extended and adapted to Simulink models. Statistical debugging utilizes an abstraction of program behavior, also known as *spectra*, (e.g., sequences of executed statements) obtained from testing. Since Simulink models have multiple outputs each of which can be tested and evaluated independently, we relate each individual Simulink model spectrum to *a test case and an output* as opposed to the existing work where each spectrum is related to *a test case*. We refer to each Simulink model spectrum as a *test execution slice*.

Given a test case that is executed on a Simulink model with multiple outputs, a *test execution slice* (or an *execution slice* for short) is a set of (atomic) blocks that were executed by the test case to generate each output. For example, Table I shows the statistical debugging results obtained based on a test suite containing six test cases (i.e., tc_1 to tc_6) for the Simulink model in Figure 1. The left-most column of Table I lists the

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

TABLE I: Test execution slices and ranking results for Simulink model in Figure 1. * denotes the faulty blocks and ✓denotes the executed blocks.

blocks in the model in Figure 1. The two columns below each test case tc_1 to tc_6 represent the two test execution slices related to each test case. Since the model in Figure 1 has two outputs, each test case execution generates two execution slices (one for `pOut` and one for `TOut`). We specify the blocks that are included in each test execution slice using a ✓. The last row of Table I shows whether each individual test execution slice passes (P) or fails (F). Among the 12 execution slices in Table I, seven are failing and five are passing. We denote each test execution slice by a tuple (tc, o) where tc is a test case and o is an output. For example, in Table I, the failing execution slice $(tc_5, pOut)$ contains nine blocks e.g., *LimitP*. That is, executing test case tc_5 results in execution of the nine blocks and yields a failure at output `pOut`.

After obtaining the test execution slices, we use a well-known statistical ranking formula, i.e. *Tarantula* [15], [31], to rank the Simulink blocks. Let b be a model block, and let $passed(b)$ and $failed(b)$, respectively, be the number of passing and failing execution slices that execute b. Let $totalpassed$ and $totalfailed$ represent the total number of passing and failing execution slices, respectively. Below is the *Tarantula* formula for computing the suspiciousness score of b:

$$Score(b) = \frac{\frac{failed(b)}{totalfailed}}{\frac{failed(b)}{totalfailed} + \frac{passed(b)}{totalpassed}} \quad (1)$$

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 rank" and a "max rank" 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, the two right-most columns of Table I respectively show the scores and the rank groups for the Simulink model example in Figure 1. Engineers are expected to inspect the blocks in that list starting from the most top ranked ones.

III. Motivation

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

When models contain multiple faults, statistical debugging can be imprecise. This is because the multiple faults that exist in a model may impact one another in unknown ways causing the impact of some faults to be masked by others. This may result in faulty blocks to be ranked low in the rankings obtained by statistical debugging. To improve the results of statistical debugging in the presence of multiple faults, researchers have proposed to cluster failures in such a way that the failures that are caused by the same faults are put in the same cluster [20]–[25].

Similar to the existing work [20]–[25], we propose an approach based on failure clustering for identifying faults in Simulink models with multiple faults. Our approach is, however, different from the existing techniques in terms of the notion of failures, the input and the technique used for clustering, and in the way we use clustering results to localize faults. We explain each of these distinguishing factors below:

Notion of failures. As mentioned in Section I, we associate a failure with the incorrect output of a test execution (i.e., a failing execution slice). Thus, in our work, clustering failures is equivalent to clustering failing execution slices. While in the existing techniques [20]–[24], a failure corresponds to a failing test case and clustering failures is equivalent to clustering failing test cases without regard to the particular outputs at which failures are observed.

Input for clustering failures. Some existing techniques [20], [22], [24], [25] take as input sequences of program elements executed by failing test cases, while other techniques [21]–[23] use sequences of program elements executed by both passing and failing test cases. The inputs to our approach are sequences of blocks executed by each test case for each output, i.e., all test execution slices, as well as the test input data used to generate these slices. Our intuition is that failures are more likely to have been caused by the same fault, not only if they execute similar blocks, but also when they use similar test inputs.

Techniques for clustering failures. Most existing approaches [21], [22], [24], [25] rely on *clustering techniques* (i.e., unsupervised learning techniques) where they group failures based on some similarity measure defined over the data that characterizes failures. Instead of relying on similarity measures, in this work, we use a *supervised learning technique* that can learn from failing and passing test executions to determine how to group the failures. Specifically, we use *Decision trees* [30] (see Section IV-A). A decision tree is built by partitioning the set of test execution slices such

that homogeneity is maximized across the resulting partitions, within certain constraints, in terms of passing and failing test execution slices. Decision trees also allow us to distinguish input data and execution trace characteristics that statistically determine failures.

Usage of clustering results. Similar to existing techniques [20], [22], we generate a single ranked list of most suspicious blocks per each cluster. However, instead of requiring engineers to inspect all ranked lists [20]–[22], our approach aims to select the most fault revealing ranked lists (i.e., those that rank faulty blocks higher), and requires engineers to inspect those selected ranked lists only (see Section IV-C). In our work, we assess the level of consistency of failing execution slices in clusters and we assume that the most consistent one will yield the best ranking.

Motivating example. We illustrate the benefits of our statistical debugging approach that relies on clustering and is used in a *one-at-a-time* debugging process using the faulty model example in Figure 1 that contains two faults: in blocks `pStand` and `LimitP`. Table I shows that testing this model produces seven failures, three of which are caused by the fault in `LimitP` and the rest are due to the fault in `pStand` The block ranking computed based on *Tarantula* is shown in the left-most column of Table I. In this ranking, the rank of `pStand` and `LimitP` are 12 and 14, respectively. Assuming engineers debug one fault at a time, they first locate the faulty block `pStand` by inspecting up to 12 blocks. After fixing this fault and re-applying the statistical debugging technique, engineers can locate the faulty block `LimitP` by inspecting at most three blocks. Thus, engineers need to inspect 15 blocks in total to locate both faults when they do not use clustering.

When we use our decision tree-based clustering, we obtain two clusters as follows: $Cluster_1$ consisting of the failing execution slices that are caused by the fault in `LimitP`; $Cluster_2$ consisting of the failing execution slices that are caused by the fault in `pStand`. For each cluster, we generate a ranked list of the most suspicious blocks using Tarantula. We then select the most fault revealing ranked list to be inspected by engineers. For this example, our approach selects the ranked list generated from $Cluster_1$ because it contains the most similar failing execution slices. By inspecting the ranked list from $Cluster_1$, engineers can find the faulty block `LimitP` by inspecting at most three blocks. We then re-apply our technique after fixing the fault at block `LimitP`. This time, our approach produces one cluster containing all the failing execution slices. Using the ranked list generated from this cluster, engineers can find the faulty block `pStand` by inspecting at most five blocks. Thus, engineers localize all faults by inspecting at most eight blocks which is significantly smaller than that of without clustering (i.e., 15 blocks).

IV. Proposed Approach

In this section, we present our approach to localize multiple faults in Simulink models. Our approach (shown in Figure 2) takes as input a faulty Simulink model, a test suite, and test oracles to determine the pass/fail information for each model output and for each test execution. We describe the three steps of our approach in Sections IV-A to IV-C.

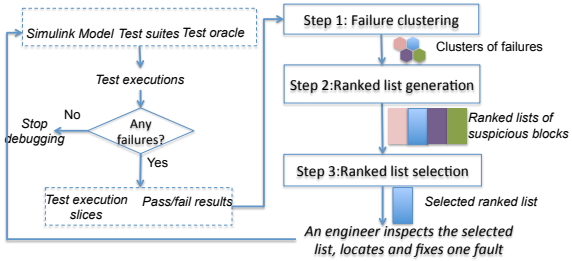


Fig. 2: Overview of our approach to identify multiple faults in Simulink models.

A. Step 1. Failure Clustering

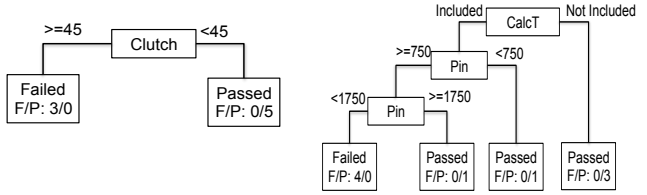
The goal of this step is to cluster failures such that the failures that are likely to have been caused by the same fault(s) are put in the same cluster. We cluster failures using *decision trees* [30], a supervised learning technique. We apply the decision tree technique to a set of test execution slices. Each test execution slice contains the following information: (a) the blocks that are covered by the test execution slice; (b) the model input variables related to each test execution slice and the values of these model input variables. Each test execution is further labeled with passing (P) and failing (F) values. Consider our model example in Figure 1. Table I shows the blocks that are covered by each execution slice, and Table II shows the model input variables and values that are used by each execution slice. For example, the execution slice $(tc_3, TOut)$ covers 18 blocks (e.g., `pAdjust`), and is generated by two model inputs, i.e., `Bypass` and `pIn` with values 5 and 1500, respectively. Further, the execution slice $(tc_3, TOut)$ is labeled with F, indicating that the execution of tc_3 results in a failure at $TOut$.

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

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

Input block	tc_1		tc_2		tc_3		tc_4		tc_5		tc_6	
	pOut	TOut	pOut	TOut	pOut	TOut	pOut	TOut	pOut	TOut	pOut	TOut
NMOT	4500	4500	6000	-	-	-	4500	4500	3000	-	6000	6000
Clutch	40	40	50	-	-	-	40	40	50	-	50	50
ByPass	20	20	20	-	5	5	20	20	20	-	20	20
Pin	-	2000	500	500	1500	1500	-	1500	-	1000	1500	1500
Tin	-	-	-	10	-	-	-	-	-	10	-	-
Pass(P)/Fail(F)	P	P	F	P	P	F	P	F	F	F	F	F

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



(a) A Decision tree for pOut. (b) A Decision tree for TOut.

Fig. 3: Decision trees generated for clustering failures at pOut and TOut.

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

In our work, for each output that fails at least once, we build a decision tree that takes as input the failing test execution slices related to that output and all the passing test execution slices. Consider our faulty model example in Figure 1. We build the two decision trees in Figures 3(a) and (b) that, respectively, relate to the failing outputs pOut and TOut. For example, the decision tree for pOut is built by using the three failing execution slices related to pOut (i.e., $(tc_2, pOut)$, $(tc_5, pOut)$), $(tc_6, pOut)$) and all the passing execution slices (i.e., $(tc1, pOut)$, $(tc1, TOut)$,$(tc2, TOut)$,$(tc3, pOut)$, $(tc4, pOut)$). Note that F/P shown in Figure 3 indicates the number of failing and passing execution slices at each leaf node. Considering the decision tree for pOut, when the value of Clutch is greater than equal to 45, the test execution slices are likely to fail at output pOut. Otherwise, these slices are likely to be passing for pOut.

Note that decision trees do not require the number of partitions to be known a priori. Instead, to build such trees, we need to have precise criteria on when to terminate the partitioning, and on how decision variables should be selected to generate new partitions at each step. Given a partition, decision trees split the partition if the partition size is not smaller than a defined threshold (i.e., *minimum split* parameter) and if splitting the partition can reasonably reduce the miss-classification error [30]. Further, decision trees rely on data homogeneity measures for selection of decision variables. In our work, we use the following termination and selection criteria to build our decision trees: We set the value of minimum split parameter to 50. This is because splitting partitions with size smaller than 50 would produce partitions with too few failing execution slices for statistical debugging to be able to distinguish faulty blocks from non-faulty ones. Moreover, we require that splitting a partition reduces the miss-classification error of decision trees by at least 1%. Finally, our decision trees use a well-known data homogeneity measure, namely *Gini Index* [30].

Having built the decision trees, we create one cluster for each partition (leaf node) if that partition contains at least one failing execution slice. Each cluster contains only the failing execution slices (and not the passing slices) of their corresponding partitions. For example, using decision trees shown in Figure 3, we obtain two clusters: $Cluster_1$= $\{(tc_2, pOut), (tc_5, pOut)), (tc_6, pOut)\}$ and $Cluster_2$ = $\{(tc3, TOut),(tc4, TOut),(tc5, TOut),(tc6, TOut)\}$.

B. Step 2. Ranked-List Generation

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

To generate a ranked list for a cluster, we compute suspiciousness scores for Simulink blocks using the *Tarantula* formula [15], [31] (see Equation 1). Specifically, the *totalfailed* corresponds to the number of failing execution slices in a given cluster and the *totalpassed* is the total number of all passing execution slices. We then rank the blocks in descending order of their suspiciousness scores. As several blocks can obtain the same score, we assign min and max ranks for each block as described in Section II.

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

C. Step 3. Ranked-List Selection

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

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

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

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

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

(2) Based on the second definition, two failing execution slices are more similar (i.e., their pairwise distance $D(S_i, S_j)$ is small), if the sets of suspicious blocks that are produced based on those slices are more similar [21], [22]. To formalize this definition, we associate to each failing execution slice S_i a vector S_i^v such that S_i^v has one element for each model block (i.e., the size of S_i^v is equal to the number of model blocks). We then create a set of slices $\mathcal{S}$ containing S_i and all the passing execution slices, and use *Tarantula* to generate a ranking $\mathcal{R}$ based on the set $\mathcal{S}$. Then we obtain the top N elements from $\mathcal{R}$. In our work, we typically set N to be 10% of the model blocks. For each element of S_i^v related to a block b, we assign one to the element if b is among the top N elements obtained from $\mathcal{R}$. Otherwise, we assign zero to that element. Note that, this way of generating a ranking $\mathcal{R}$ for a failing execution slice S_i has been first proposed in [22] where the goal was to obtain a ranking for a failing test case.

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

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

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

V. Empirical Evaluation

In this section, we describe our research questions (Section V-A), experiment settings (Section V-B), evaluation metrics (Section V-C), and experiment results (Section V-D).

A. Research Questions

RQ1. [Fault Localization Accuracy] *Can our decision tree-based clustering approach help localizing faults by ranking the faulty blocks in the top most suspicious blocks? How does the fault localization ability of our approach compare with that of the non-clustering approach and the existing clustering approaches?* We investigate the accuracy of our approach in identifying faults in Simulink models with multiple faults. Specifically, we evaluate the maximum number of blocks inspected to identify faults at different debugging iterations. We compare our results with those obtained by two alternative debugging techniques for Simulink models with multiple faults: (1) Statistical debugging without using clustering (2) Statistical debugging combined with the pairwise clustering technique. The latter is a state-of-the-art clustering technique based on statistical debugging previously proposed for identifying multiple faults in source code [22]. We implemented and adapted this technique for Simulink models to use it as a baseline clustering technique for comparison with our work.

RQ2. [Fault Localization Cost] *Can our decision tree-based clustering approach significantly lower the cost of identifying*

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

RQ3. [Robustness] *Does the fault localization ability of our approach remain robust when it is applied to models with an unknown (and potentially large) number of faults? How does our approach compare with the pairwise clustering and the non-clustering approaches in terms of robustness?* In order for our approach to be effective in localizing multiple faults, its fault localization ability should remain robust (i.e., show a graceful degradation) when the number of faults in the model grows. We study the changes in the fault localization ability of our approach when applied to models containing different numbers of faults and compare those changes with the changes in the fault localization ability of the non-clustering and the pairwise clustering approaches applied to the same models.

B. Experiment Settings

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

Industrial Subjects. We use three Simulink models developed by Delphi Automotive in our experiments. We refer to these three models as *MS*, *MC*, and *MGL*. Table III shows the number of subsystems, atomic blocks, links, and inputs and outputs of each model. Note that the models that we chose are representative in terms of size and complexity among the Simulink models developed at Delphi. Further, these models include about ten times more blocks than the publicly available Simulink models from the Mathworks model repository [33].

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

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

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

TABLE III: Key information about industrial subjects.

Test suite and test oracle. We generated three test suites, each of which with 200 test cases for *MS*, *MC*, and *MGL* using Adaptive Random Testing [34]. Adaptive random testing is a black box and lightweight test generation strategy that distributes test cases evenly within valid input ranges, and thus, helps ensure diversity among test cases. Note that the size of the test suites was based on typical practice at Delphi given test budget constraints and the oracle costs. Further, we used the fault-free versions of our industrial subjects for test oracles.

Experiment design. To answer our research questions, we applied our approach (in a one-at-a-time debugging process) on our 240 faulty models. The number of debugging iterations for each faulty model is at most equal to the number of faults activated for that faulty model. This is because, at each iteration, we resolve all the faults located in the same rank as that of the top most ranked faulty block. For each faulty model and at each iteration, we run our approach outlined in Figure 2 by applying a test suite with 200 test cases to obtain test execution slices. We then subsequently apply the three steps in Figure 2 to generate a selected ranked list of suspicious blocks which is used by engineers to find one fault. We inspect the ranking manually to identify the first faulty block and we remove that fault by deactivating its corresponding switch. We then re-iterate the approach in Figure 2 until all faults are removed. We repeated the above experiment for the 240 faulty models twice: One time for the QC_{Trace} selection criterion, and the second time for the QC_{Rank} selection criterion. We denote our decision tree-based clustering approach that uses QC_{Trace} and QC_{Rank} by *DT-QCTrace* and *DT-QCRank*, respectively.

As specified in the research questions, in our experiment, we consider two baseline techniques for comparison: A traditional statistical debugging technique without clustering (denoted by *NC*), and a statistical multi-fault debugging approach [22] that uses a *pairwise* clustering technique that we adapted to Simulink models. We repeated the above experiment for the *NC* and the *pairwise* approaches. Our implementation of the pairwise approach uses the setting used by Jones et al. [22] except that we consider the top 10% of the blocks to build the clusters as opposed to the top 20%. This is because in our previous work [19], we have shown that the top 10% of the Simulink blocks in a ranking are likely to contain most faults. As for the pairwise approach, since several clusters are generated, we use our two selection criteria discussed in Section IV-C to select a ranking with the highest fault revealing ability. Specifically, we denote the pairwise approach that uses QC_{Trace} and QC_{Rank} by *PW-QCTrace* and *PW-QCRank*, respectively. In summary, we repeated our experiment 682, 679, 621, 673, and 659 times for *DT-QCTrace*, *DT-QCRank*,

PW-QCTrace, *PW-QCRank*, and *NC*, respectively. Note that, at each iteration, we resolve all the faults located in the same rank as that of the top most ranked faulty block. Thus, different fault localization techniques require different numbers of iterations to resolve all the faults. We ran our experiment on a high performance computing platform with 2 clusters, 280 nodes, and 3904 cores. Our experiment were executed on different nodes of a cluster with Intel Xeon L5640@2.26GHz processor. The total computation time for our experiment (using a single node) is 15548 hours. Most of the experiment time is used to generate test execution slices. In total, we generated 1744000, 1503600, and 3000400 test execution slices for *MS*, *MC*, and *MGL*, respectively.

C. Evaluation Metrics

Since we experiment with faulty models with multiple faults and fault localization is applied iteratively until models are fault-free, we provide two new metrics: *maximum rank of faulty blocks* and *fault localization cost*. In our work, at each iteration, we identify the faulty block that is ranked highest in the ranked list generated at that iteration. For each identified faulty block, the *maximum rank of faulty blocks* is the max rank of the rank group containing that faulty block. The *fault localization cost* is the total number of blocks that need to be inspected to localize all the faults in a given faulty model over all the iterations and making the model fault-free. Note that when several faults are in the same rank group in a ranked list, we assume that all of them are localized when engineers inspect that ranked list. Thus, the total iterations for obtaining a fault-free model can be smaller than the number of faults in that model. Note that the *fault localization cost* is an adaptation of the *absolute number of blocks inspected* metric used in the literature for *single-fault* localization in code [11], [12], [14], [17], [31], [35].

D. Experiment Results

1) RQ 1. Fault Localization Accuracy: To answer this question, we compute the number of faults and the proportion of faults that are ranked among the top blocks in some ranked list generated at some iteration by each of the *DT-QCTrace*, *DT-QCRank*, *PW-QCTrace*, *PW-QCRank*, and *NC* fault localization techniques. Figure 4 shows the number of faults that are ranked among the top blocks when these techniques are applied to our 240 faulty versions containing, in total, 840 faults. In this figure, the X-axis shows the number of top N ($N = \{10, 20, ..., 200\}$) blocks and the Y-axis shows the number of faults located among the top N blocks at some rank list produced at some fault localization iteration by each of the above five techniques. Based on Figure 4, when we use *DT-QCTrace* and *DT-QCRank*, 95 out of the total of 840 faults are ranked among the top 10 blocks in some ranked list at some iteration. In contrast, by using *NC*, *PW-QCTrace* and *PW-QCRank*, 23, 82, and 86 faults are ranked among the top 10 blocks at some iteration, respectively. In general, *DT-QCTrace* is able to rank more faults among the top ranked blocks compared to the other four techniques. After *DT-QCTrace*, *DT-QCRank* is the best. Further, both *DT-QCTrace* and *DT-QCRank* are better than *PW-QCTrace* and *PW-QCRank*. *NC* is worse than *PW-QCTrace* and *PW-QCRank* when we are interested in faults ranked among the top 50 blocks.

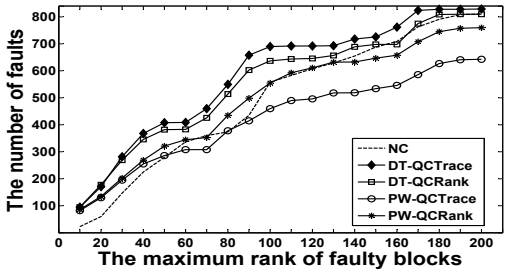
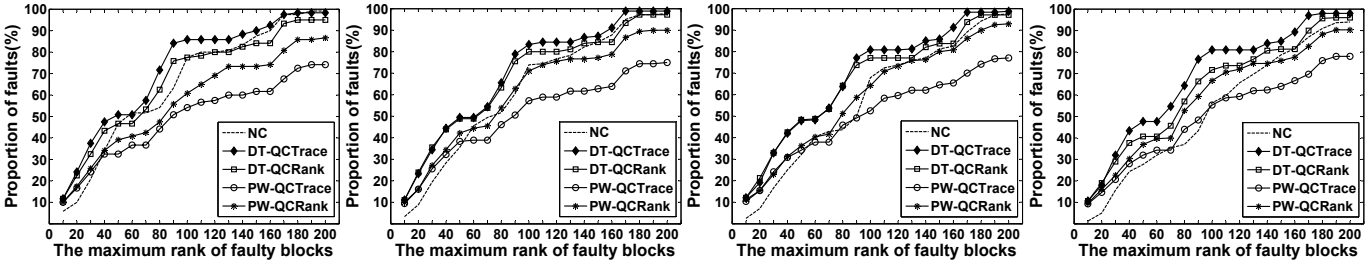


Fig. 4: The number of faults vs. the maximum rank of faulty blocks for all the 840 faults.

Figures 5(a) to (d) show the proportion of faults that are ranked among the top blocks when our five fault localization techniques are, respectively, applied to the 120 faults seeded into our two-fault models, the 180 faults seeded into our three-fault models, the 240 faults seeded into our four-fault models, and the 300 faults seeded into our five-fault models, respectively. Note in Figures 5(a) to (d), the X-axis is the same as the X-axis in Figure 4, but the Y-axis shows the proportion (instead of the absolute number) of faults ranked high, because the numbers of faults seeded into two-fault to five-fault models are different form one another. Based on Figure 5(a), using *DT-QCTrace*, 61 out of the 120 faults (i.e., more than 50% of the faults) seeded into the two-fault models are ranked among the top 50 blocks. In contrast, using *DT-QCRank*, *NC*, *PW-QCTrace*, and *PW-QCRank*, the 61 faults are ranked among the top most 70, 60, 90, and 90 blocks, respectively.

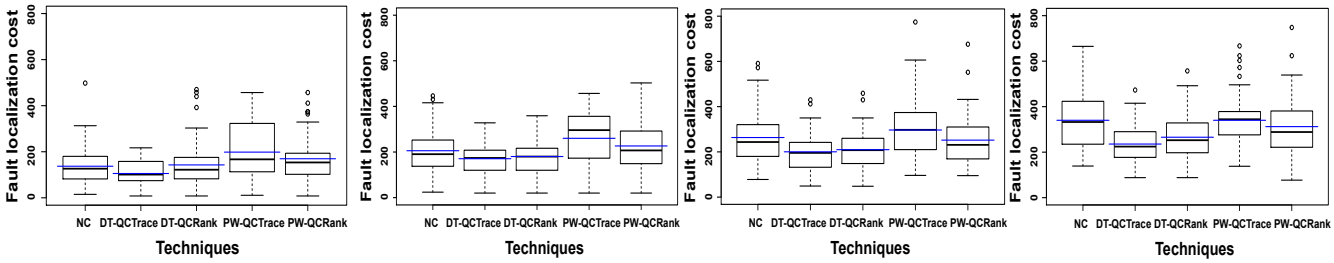
In general, the results in Figure 5 show that *DT-QCTrace* and *DT-QCRank* always perform better than *PW-QCTrace* and *PW-QCRank*. Further, *DT-QCTrace* always performs better than *NC* for three-fault to five-fault models, and also for two-fault models when we consider the faults that are ranked among the top 60 blocks. *DT-QCRank* always performs better than *NC* when we consider the faults that are ranked among the top 50 blocks. Note that it is expected for *NC* to eventually converge to the same performance as that of our clustering technique (*DT-QCTrace* and *DT-QCRank*) when the number of faults in our models are small (e.g., two-fault models). This is because faults are less likely to mask one another, and hence, the rankings generated by *NC* are less impacted when the number of faults are small (e.g. two faults).

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



(a) Two-fault models. (b) Three-fault models. (c) Four-fault models. (d) Five-fault models.

Fig. 5: Proportion of faults vs. the maximum rank of faulty blocks for two-fault to five-fault models.



(a) Two-fault models. (b) Three-fault models. (c) Four-fault models. (d) Five-fault models.

Fig. 6: Distributions of fault localization cost for two-fault to five-fault models.

2) RQ2. Fault Localization Cost: To answer this question, we compute the fault localization cost values for all the 240 faulty models and for each of the *DT-QCTrace*, *DT-QCRank*, *PW-QCTrace*, *PW-QCRank*, and *NC* fault localization techniques. Figures 6(a) to (d) show the distributions of the fault localization cost values for our two-fault to five-fault models, respectively. Specifically, each box-plot consists of 60 points corresponding to the 60 faulty versions in each of the two-fault to the five-fault model groups. In each of these figures, the X-axis shows the five fault localization techniques, and the Y-axis shows the fault localization cost.

To statistically compare the fault revealing ability of different fault localization techniques, we performed the *non-parametric pairwise Wilcoxon Pairs Signed Ranks test* [36], and calculated the effect size using *Cohen's* d [37]. The level of significance (α) was set to 0.05, and, following standard practice, d was labeled "small" for $0.2 \leq d < 0.5$, "medium" for $0.5 \leq d < 0.8$, and "high" for $d \geq 0.8$ [37].

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

In summary, the answer to **RQ2** is that our decision tree-based techniques significantly improve the fault localization cost compared to *NC*, *PW-QCTrace*, and *PW-QCRank*. Further, on average, *DT-QCTrace* reduces the fault localization cost by 59 blocks (25%) compared to *NC*, and by 62 blocks (26%) compared to *PW-QCRank*.

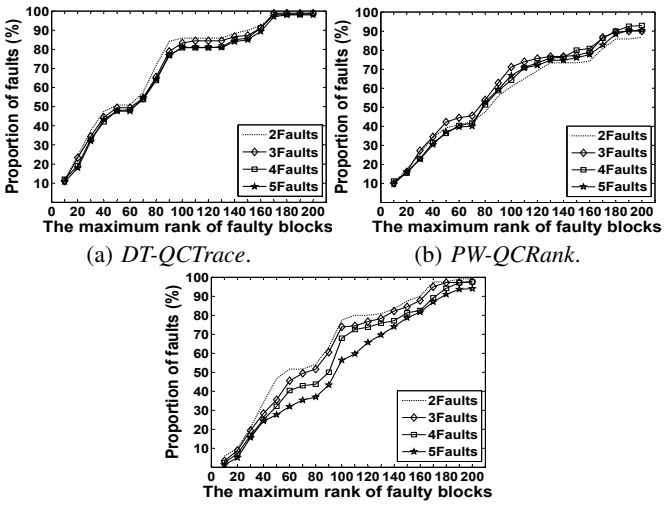
3) RQ3. Robustness: For this question, we consider *DT-QCTrace*, *PW-QCRank* and *NC* because based on our results in **RQ1** and **RQ2**, *DT-QCRank*, *PW-QCTrace* underperform *DT-QCTrace* and *PW-QCRank*, respectively. To answer this question, we evaluated the changes in the proportion of faults that are ranked among the top blocks as we vary the number of faults seeded in the underlying faulty models. Figures 7(a) to (c) show the results for *DT-QCTrace*, *PW-QCRank*, and *NC*, respectively. In each figure, we show how the performance of each of these three techniques is impacted when that technique is applied to the two-fault, the three-fault, the four-fault, and the five-fault models separately.

The data in Figure 7 was already shown in Figure 5 where we showed that *DT-QCTrace* outperforms other techniques. Figure 7, however, compares the robustness of these techniques as the number of faults changes. As this figure shows, *DT-QCTrace* is the most robust technique since its performance changes the least as the number of faults increases from two to five. The maximum deviation for *DT-QCTrace* is 7.9% and the average deviation is 3.7%. *PW-QCRank* is less robust than *DT-QCTrace* but more robust than *NC* with maximum and average deviations of 10.3% and 5.5%, respectively. Finally, *NC* is the least robust technique with maximum and average deviations of 21.2% and 11.9%, respectively.

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

VI. Related Work

In this section, we compare our work with the existing fault localization techniques that aim to localize multiple faults in programs [16], [20]–[29], [38], [39].



(a) *DT-QCTrace*. (b) *PW-QCRank*.

(c) *NC*.

Fig. 7: Proportions of faults (y-axis) among the top-N ranks (x-axis) obtained by *DT-QCTrace*, *PW-QCRank*, and *NC*.

Several techniques [20]–[22], [29] cluster failures and require engineers to inspect either failures in all the clusters or all rankings obtained from the clusters. Multidimensional scaling has been used for clustering failures based on similarities between execution profiles in [29] and between statistical rankings in [21]. Jones et al. [22] cluster failures using hierarchical clustering and pairwise clustering. They, further, propose a parallel debugging process to inspect all the ranked lists obtained from all the clusters. Steimann et al. [20] use integer linear programming to cluster failures and generate statistical rankings for the clusters. These approaches have not been evaluated nor adapted to one-at-a-time debugging. Hence, they require engineers to inspect all the clusters (or all the ranked lists) at once, potentially missing the masked faults or wasting effort by identifying the same faults more than once. Further, none of these techniques use both input values and execution slices (traces) as the input for clustering. Our work uses a supervised learning technique applied to heterogeneous input data, and is designed and evaluated for a one-at-a-time debugging process, matching how Simulink models are often debugged in practice.

Instead of generating several rankings, the following techniques generate one ranking. Zheng et al. [24] cluster failing test executions and predicates. Jiang and Su [25] cluster predicates using a k-mean technique, identify the most predictive predicates, and generate one ranking in terms of a control-flow graph. Abreu et al. [26], [27] combine statistical debugging with logical reasoning to rank sets of program elements. Brun and Ernst [40] build predictor models based on program revisions to produce a subset of program properties that might be faulty. Cellier et al. [38] cluster failures using association rules to obtain a single ranking and propose a mechanism based on formal concept analysis to guide ranking inspection. These techniques aim to find multiple faults using a single ranking. Only the work in [38] provides guidelines on when an inspection can be stopped, but no evaluation is reported. In contrast, our approach aims to identify one ranking in which at least one fault is top-ranked. Further, our approach is iterative, so that faults that are ranked low in the first iterations, can be ranked higher in subsequent iterations. Finally, our work is evaluated using industrial case studies.

Liblit et al. [16], [28] iteratively re-rank predicates using only the execution traces that do not execute the top-ranked predicates identified in the previous iterations. As they do not re-generate execution traces after fixing faults, the masked faults may remain undetected. In our earlier work [23], we have also used decision trees based on input equivalence classes to cluster failures in the context of black-box testing. Our current work uses both execution traces and test inputs for clustering. Further, we assume a one-at-a-time debugging process and select one ranking per iteration, while in [23], clusters are combined together to generate a single ranking and the approach is not iterative. Finally, we have applied our technique to industrial Simulink models with multiple faults. None of the above have been applied to Simulink models.

VII. Conclusion

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

In our paper, we studied these fault types: wrong function, wrong value, and wrong connection. In future, we plan to consider other fault types, e.g., missing blocks or missing connections. Further, we plan to localize multiple faults in Stateflow [41] (state machine) models.

Acknowledgments

Supported by the Fonds National de la Recherche, Luxembourg (FNR/P10/03 - Verification and Validation Laboratory, and FNR8003491), and Delphi Automotive Systems, Luxembourg.

REFERENCES

[1] R. Reicherdt and S. Glesner, "Slicing matlab simulink models," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12, 2012, pp. 551–561.

[2] A. Sridhar and D. Srinivasulu, "Slicing matlab simulink/stateflow models," in *Intelligent Computing, Networking, and Informatics*. Springer, 2014, pp. 737–743.

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

[4] A. Thums and J. Quante, "Reengineering embedded automotive software," in *Proceedings of the 28th IEEE International Conference on Software Maintenance*, ser. ICSM '12, 2012, pp. 493–502.

[5] MathWorks, "Simulink," http://www.mathworks.nl/products/simulink/.

[6] T. Ball, M. Naik, and S. 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*, ser. POPL '03, 2003, pp. 97–105.

[7] H. Cleve and A. Zeller, "Finding failure causes through automated testing," in *Proceedings of the Fourth International Workshop on Automated Debugging*, ser. AADEBUG 00, 2000.

[8] A. Groce, D. Kroening, and F. Lerda, "Understanding counterexamples with explain," in *Computer Aided Verification*. Springer, 2004, pp. 453–456.

[9] R. Hildebrandt and A. Zeller, "Simplifying failure-inducing input," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '00, 2000, pp. 135–145.

[10] A. Orso, J. A. Jones, M. J. Harrold, and J. T. Stasko, "Gammatella: Visualization of program-execution data for deployed software," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04, 2004, pp. 699–700.

[11] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 20th International Symposium on Software Testing and Analysis*, ser. ISSTA '11, 2011, pp. 199–209.

[12] M. Renieris and S. P. Reiss, "Fault localization with nearest neighbor queries," in *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, ser. ASE '03, 2003, pp. 30–39.

[13] X. Zhang, R. Gupta, and Y. Zhang, "Precise dynamic slicing algorithms," in *Proceedings of the 25th International Conference on Software Engineering*, ser. ICSE '03, 2003, pp. 319–329.

[14] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. ACM, 2005, pp. 342–351.

[15] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02, 2002, pp. 467–477.

[16] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 15–26, 2005.

[17] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "Sober: statistical model-based bug localization," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 286–295, 2005.

[18] R. Abreu, P. Zoeteweij, and A. J. 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*. IEEE, 2007, pp. 89–98.

[19] B. Liu, Lucia, S. Nejati, and L. Briand, "Simulink fault localization: an iterative statistical debugging approach," *Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Tech. Rep.TR-SnT-2015-8*, 2015.

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

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

[22] J. A. Jones, J. F. Bowring, and M. J. Harrold, "Debugging in parallel," in *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM, 2007, pp. 16–26.

[23] L. C. Briand, Y. Labiche, and X. Liu, "Using machine learning to support debugging with tarantula," in *Software Reliability, 2007. ISSRE'07. The 18th IEEE International Symposium on*. IEEE, 2007, pp. 137–146.

[24] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical debugging: simultaneous identification of multiple bugs," in *Proceedings of the 23rd international conference on Machine learning*. ACM, 2006, pp. 1105–1112.

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

[26] R. Abreu, P. Zoeteweij, and A. Gemund, "Localizing software faults simultaneously," in *Quality Software, 2009. QSIC'09. 9th International Conference on*. IEEE, 2009, pp. 367–376.

[27] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "Spectrum-based multiple fault localization," in *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*. IEEE, 2009, pp. 88–99.

[28] P. Arumuga Nainar and B. Liblit, "Adaptive bug isolation," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 255–264.

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

[30] L. Olshen, C. J. Stone *et al.*, "Classification and regression trees," *Wadsworth International Group*, vol. 93, no. 99, p. 101, 1984.

[31] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05, 2005, pp. 273–282.

[32] F. Steimann, M. Frenkel, and R. Abreu, "Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 2013, pp. 314–324.

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

[34] T. Y. Chen, H. Leung, and I. Mak, "Adaptive random testing," in *Advances in Computer Science-ASIAN 2004. Higher-Level Decision Making*. Springer, 2005, pp. 320–329.

[35] Lucia, D. Lo, and X. Xia, "Fusion fault localizers," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14, 2014, pp. 127–138.

[36] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics bulletin*, pp. 80–83, 1945.

[37] R. J. Grissom and J. J. Kim, "Effect sizes for research," *A broad practical approach. Mah*, 2005.

[38] P. Cellier, M. Ducassé, S. Ferré, and O. Ridoux, "Multiple fault localization with data mining." in *SEKE*, 2011, pp. 238–243.

[39] J. Röβler, G. Fraser, A. Zeller, and A. Orso, "Isolating failure causes through test case generation," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012, pp. 309–319.

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

[41] MathWorks, "Stateflow," http://www.mathworks.nl/products/stateflow/.