Improving Fault Localization for Simulink Models using Search-Based Testing and Prediction Models

Bing Liu, Lucia, Shiva Nejati, Lionel Briand
SnT Centre, University of Luxembourg, Luxembourg
Email: {bing.liu, lucia.lucia, shiva.nejati, lionel.briand}@uni.lu

***Abstract*—One promising way to improve the accuracy of fault localization based on statistical debugging is to increase diversity among test cases in the underlying test suite. In many practical situations, adding test cases is not a cost-free option because test oracles are developed manually or running test cases is expensive. Hence, we require to have test suites that are both *diverse* and *small* to improve debugging. In this paper, we focus on improving fault localization of Simulink models by generating test cases. We identify three test objectives that aim to increase test suite diversity. We use these objectives in a search-based algorithm to generate diversified but small test suites. To further minimize test suite sizes, we develop a prediction model to stop test generation when adding test cases is unlikely to improve fault localization. We evaluate our approach using three industrial subjects. Our results show (1) the three selected test objectives are able to significantly improve the accuracy of fault localization for small test suite sizes, and (2) our prediction model is able to maintain almost the same fault localization accuracy while reducing the average number of newly generated test cases by more than half.**

***Index Terms*—Fault localization, Simulink models, search-based testing, test suite diversity, and supervised learning.**

I. Introduction

The embedded software industry increasingly relies on model-based development methods to develop software components [1]. These components are largely developed in the Matlab/Simulink language [2]. An important reason for increasing adoption of Simulink in embedded domain is that Simulink models are executable and facilitate *model testing* or *simulation*, (i.e., design time testing based on system models) [3], [4]. To be able to identify early design errors through Simulink model testing, engineers require effective debugging and fault localization strategies for Simulink models.

Statistical debugging is a lightweight and well-studied debugging technique [5]–[12]. Statistical debugging localizes faults by ranking program elements based on their suspiciousness scores. These scores capture faultiness likelihood for each element and are computed based on statistical formulas applied to sequences of executed program elements (i.e., spectra) obtained from testing. Developers use such ranked program elements to localize faults in their code.

In our previous work [13], we extended statistical debugging to Simulink models and evaluated the effectiveness of statistical debugging to localize faults in Simulink models. Our approach builds on a combination of statistical debugging and dynamic slicing of Simulink models. We showed that the accuracy of our approach, when applied to Simulink models from the automotive industry, is comparable to the accuracy of statistical debugging applied to source code [13]. We further extended our approach to handle fault localization for Simulink models with multiple faults [14].

Since statistical debugging is essentially heuristic, despite various research advancements, it still remains largely unpredictable [15]. In practice, it is likely that several elements have the same suspiciousness score as that of the faulty, and hence, be assigned the same rank. Engineers will then need to inspect all the elements in the same rank group to identify the faulty element. Given the way statistical debugging works, if every test case in the test suite used for debugging executes either both or neither of a pair of elements, then those elements will have the same suspiciousness scores (i.e., they will be put in the same rank group). One promising strategy to improve precision of statistical debugging is to use an existing ranking to generate additional test cases that help *refine* the ranking by reducing the size of rank groups in the ranking [15]–[18].

In situations where test oracles are developed manually or when running test cases is expensive, adding test cases is not a zero-cost activity. Therefore, an important question, which is less studied in the literature, is how we can refine statistical rankings by generating a *small* number of additional test cases? In this paper, we aim to answer this question for fault localization of Simulink models. While our approach is not particularly tied to any modeling or programming language, we apply our work to Simulink since, in some domains (e.g., automotive), it is expensive to execute Simulink models and to characterize their expected behaviour [4], [19]. This is because Simulink models include computationally expensive physical models [20], and their outputs are complex continuous signals [19]. We identify three alternative test objectives that aim to generate test cases exercising diverse parts of the underlying code and adapt these objectives to Simulink models [15], [16], [21]. We use these objectives to develop a search-based test generation algorithm, which builds on the whole test suite generation algorithm [22], to extend an existing test suite with a small number of test cases. Given the heuristic nature of statistical debugging, adding test cases may not necessarily improve fault localization accuracy. Hence, we use the following two-step strategy to stop test generation when it is unlikely to be worthwhile: First, we identify Simulink *super blocks* through static analysis of Simulink models. Given a Simulink model M, a super block is a set B of blocks of M such that, for any test case tc, tc executes either *all* or *none* of the blocks in B. That is,

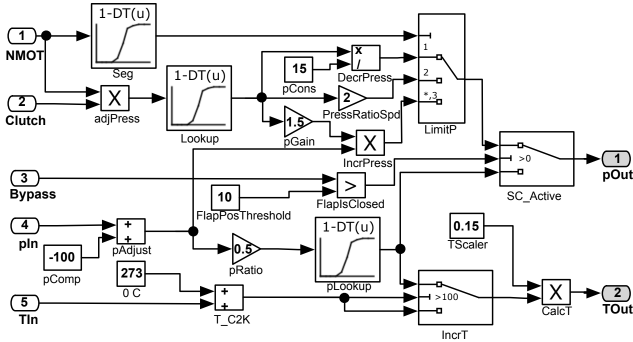

Fig. 1: A Simulink model example where `pRatio` is faulty.

there is no test case that executes a subset (and not all) of the blocks in a super block. Statistical debugging, by definition, always ranks the blocks inside a super block together in the same rank group. Thus, when elements in a rank group are all from a super block, the rank group cannot be further refined through statistical debugging, and hence, test generation is not beneficial. Second, we develop a prediction model based on supervised learning techniques, specifically decision trees [23] using historical data obtained from previous applications of statistical debugging. Our prediction model effectively learns rules that relate improvements in fault localization accuracies to changes in statistical rankings obtained before and after adding test cases. Having these rules and having a pair of statistical rankings from before and after adding some test cases, we can predict whether test generation should be stopped or continued. *Our Contributions include:*

- We develop a search-based testing technique for Simulink models that uses the existing alternative test objectives [15], [16], [21] to generate small and diverse test suites that can help improve fault localization accuracy.

- We develop a strategy to stop test generation when test generation is unlikely to improve fault localization. Our strategy builds on static analysis of Simulink models and prediction models built based on supervised learning.

- We have evaluated our approach using three industrial subjects. Our experiments show that: **(i)** The three alternative test objectives are equally capable of improving the accuracy of fault localization for Simulink models and with small test suite sizes, and they are able to produce an accuracy improvement that is statistically higher than the improvement obtained by random test generation (baseline). **(ii)** Our strategy based on static analysis and supervised learning is able to stop generating test cases that are not beneficial for fault localization. In particular, on average, by generating only 11 test cases, we are able to obtain an accuracy improvement close to that obtained by 25 test cases when our strategy to stop test generation is not used.

II. Background and Notation

In this section, we provide some background and fix our formal notation. Figure 1 shows an example of a Simulink model. This model takes five input signals and produces two output signals. It contains 21 Simulink (atomic) blocks. Simulink blocks are connected via lines that indicate data

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

Block Name	t_1		t_2		t_3		t_4		Score	Rank (Min-Max)
	pOut	TOut	pOut	TOut	pOut	TOut	pOut	TOut		
SC_Active	✓		✓		✓		✓		0	-
FlapIsClosed	✓		✓		✓		✓		0	-
FlapPosThreshold	✓		✓		✓		✓		0	-
LimitP	✓						✓		0	-
Seg	✓						✓		0	-
adjPress	✓						✓		0	-
Lookup	✓						✓		0	-
DecrPress	✓								0	-
pCons	✓								0	-
PressRatioSpd							✓		0	-
IncrPress									NaN	-
pGain									NaN	-
pRatio*		✓	✓		✓	✓			**0.7**	**1-9**
pLookup		✓	✓		✓	✓			0.7	1-9
pComp		✓	✓		✓	✓			0.7	1-9
pAdjust		✓	✓		✓	✓			0.7	1-9
CalcT		✓		✓		✓		✓	0.7	1-9
Tscaler		✓		✓		✓		✓	0.7	1-9
IncrT		✓		✓		✓		✓	0.7	1-9
T_C2K		✓		✓		✓		✓	0.7	1-9
0 C		✓		✓		✓		✓	0.7	1-9
Pass(P)/Fail(F)	P	F	P	P	P	P	P	P		

flow connections. Formally, a Simulink model is a tuple $(Nodes, Links, Inputs, Outputs)$ where $Nodes$ is a set of Simulink blocks, $Links \subseteq (Nodes \times Nodes)$ is a set of links between the blocks, $Inputs$ is a set of input ports and $Outputs$ is a set of output ports.

In our previous work [13], 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, we relate each individual Simulink model spectrum to *a test case and an output*. We refer to each Simulink model spectrum as a *test execution slice*. A *test execution slice* is a set of (atomic) blocks that were executed by a test case to generate each output [13].

Let TS be a test suite. Given each test case $tc \in TS$ and each output $o \in Outputs$, we refer to the set of Simulink blocks executed by tc to compute o as a *test execution slice* and denote it by $tes_{tc,o}$. Formally, we define $tes_{tc,o}$ as follows.

$$tes_{tc,o} = \{n \mid n \in static_slice(o) \wedge tc \text{ executes } n \text{ } for \text{ } o\}$$

where $static_slice(o) \subseteq Nodes$ is the *static backward slice* of output o and is equal to the set of all nodes in $Nodes$ that can reach output o via data or control dependencies. We denote the set of all test execution slices obtained by a test suite TS by TES_{TS}. In [13], we provided a detailed discussion on how the static backward slices (i.e., $static_slice(o)$) and test execution slices (i.e., $tes_{tc,o}$) can be computed for Simulink models.

For example, suppose we seed a fault into the model example in Figure 1. Specifically, we change the constant value used by the gain block (`pRatio`) from 0.65 to 0.5, i.e., the input of `pRatio` is multiplied by 0.5 instead of 0.65. Table I shows the list of blocks in this model and eight test execution slices obtained from running four test cases (i.e., tc_1 to tc_4) on this model. In this example, each test case generates two execution slices (one for each model output). 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).

After obtaining the test execution slices, we use a well-known statistical ranking formula, i.e. *Tarantula* [6], [24],

to rank the Simulink blocks. Note that our comparison [13] of alternative statistical formulas applied to Simulink models revealed no significant difference among these formulas and Tarantula. 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:

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

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*. For each rank group, we assign a "min rank" and a "max rank". The min (respectively, max) rank indicates the least (respectively, the greatest) numbers of blocks that need to be inspected if the faulty block happens to be in this group. For example, Table I shows the scores and the rank groups for our example in Figure 1. Based on this table, engineers may need to inspect at least one block and at most nine blocks in order to locate the faulty block `pRatio`.

III. Test Generation for Fault Localization

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

A. Search-based Test Generation

We use *search-based* techniques [25] to generate test cases that improve statistical debugging results. To guide the search algorithm, we define fitness functions that aim to increase diversity of test cases. Our intuition is that diversified test cases are likely to execute varying subsets of Simulink model blocks. As a result, Simulink blocks are likely to take different scores, and hence, the resulting rank groups in the statistical ranking are likely to be smaller. In this section, we first present the fitness functions that are used to guide test generation, and then, we discuss the search-based test generation algorithm. We describe three different alternative fitness functions referred to as *coverage dissimilarity*, *coverage density* and *number of dynamic basic blocks*. Coverage dissimilarity has previously been used for test prioritization [21], and is used in this paper for the first time to improve fault localization. The two other alternatives, i.e., *coverage density* [15] and *number of dynamic basic blocks* [16], have been previously used to improve source code fault localization.

SimulinkFaultLocalization()
Input: - TS: An initial test suite
- M : A Simulink model
- $round$: The number of test generation rounds
- k: The number of new test cases per round

Output: $rankList$: A statistical debugging ranking

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

Fig. 2: Overview of our Simulink fault localization approach.

Coverage Dissimilarity. Coverage dissimilarity aims to increase diversity between test execution slices generated by test cases. We use a set-based distance metric known as Jaccard distance [26] to define coverage dissimilarity. Given a pair $tes_{tc,o}$ and $tes_{tc',o'}$ of test execution slices, we denote their dissimilarity as $d(tes_{tc,o}, tes_{tc',o'})$ and define it as follows:

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

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

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

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

Coverage Density. Campos et al [15] argue that the accuracy of statistical fault localization relies on the density of test coverage results. They compute the test coverage density as the average percentage of components covered by test cases over the total number of components in the underlying program. We adapt this computation to Simulink, and compute the coverage density of a test suite TS, denoted by $p(TS)$, as follows:

$$\text{p}(TS) = \frac{1}{|TES_{TS}|} \sum_{tes_{tc,o} \in TES_{TS}} \frac{|tes_{tc,o}|}{|static_slice(o)|}$$

That is, our adaptation of coverage density to Simulink computes, for every output o, the average size of test execution slices related to o over the static backward slice of o. Note that a test execution slice related to output o is always a subset of the static backward slice of o. Low values of $p(TS)$ (i.e., close to zero) indicate that test cases cover small parts of the underlying model, and high values (i.e., close to one) indicate that test cases tend to cover most parts of the model. According to Campos et al [15], a test suite whose coverage density is equal to 0.5 (i.e., neither low nor high) is more capable of generating accurate statistical ranking results. Similar to Campos et al [15], we define the coverage density fitness function as $fit_{Dens}(TS) = |0.5 - p(TS)|$ and aim to minimize $fit_{Dens}(TS)$ to obtain more accurate ranking results.

Number of Dynamic Basic Blocks. Given a test suite TS for fault localization, a *Dynamic Basic Block (DBB)* [16] is a subset of program statements such that for every test case $tc \in TS$, all the statements in *DBB* are either all executed together by tc or none of them is executed by tc. According to [16], a test suite that can partition the set of statements of the program under analysis into a large number of dynamic basic blocks is likely to be more effective for statistical debugging. In our work, we (re)define the notion of DBB for Simulink models based on test execution slices. Formally, a set *DBB* is a dynamic basic block iff $DBB \subseteq Nodes$ and for every test execution slice $tes \in TES_{TS}$, we have either $DBB \subseteq tes$ or $DBB \cap tes = \emptyset$. For a given set TES_{TS} of test execution slices obtained by test suite TS, we can partition the set $Nodes$ of Simulink model blocks into a number of *disjoint* dynamic basic blocks $DBB_1, \ldots, DBB_l$. Our third fitness function, which is defined based on dynamic basic blocks and is denoted by $fit_{dbb}(TS)$, is defined as the number of dynamic basic blocks produced by a given test suite TS, i.e., $fit_{dbb}(TS) = l$. The larger the number dynamic basic blocks, the better the quality of a test suite TS for statistical debugging. For example, the test suite in Table I partitions the model blocks in Figure 1 into six DBBs. An example DBB for that model includes the following blocks: `CalcT`, `TScaler`,`IncrT`, `T_C2K`, `0 C`.

Test generation algorithm. Having defined the fitness functions, we now define our search-based test generation algorithm (i.e. TESTGENERATION in Figure 2). The TESTGENERATION algorithm is shown in Figure 3 and generates new test cases based on any of our three fitness functions. The algorithm adapts a single-state search optimizer [25]. In particular, it builds on the Hill-Climbing with Random Restarts (HCRR) algorithm [25]. We chose to build on HCRR because, in our previous work on testing Simulink models [27], HCRR was able to produce the best optimized test cases among other single-state optimization algorithms. Computation of all the three fitnesses we described earlier rely on test execution slices. To obtain test execution slices, we need to execute test cases on Simulink models. This makes our fitness computation expensive. Hence, in this paper, we rely on single-state search optimizers as opposed to population-based search techniques.

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

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

Fig. 3: Test case generation algorithm.

The algorithm in Figure 3 receives as input the existing set of test execution slices TES_{TS}, a Simulink model M, and the number of new test cases that need to be generated (k). The output is a test suite ($newTS$) of k new test cases. The algorithm starts by generating an initial randomly generated set of k test cases TS_{curr} (Line 1). Then, it computes the fitness of TS_{curr} (Line 3) and sets TS_{curr} as the current best solution (Line 4). The algorithm then searches for a best solution through two nested loops: (1) *The internal loop* (Lines 6 to 12). This loop tries to find an optimized solution by locally tweaking the existing solution. That is, the search in the inner loop is *exploitative*. The mutation operator in the inner loop generates a new test suite by tweaking the individual test cases in the current test suite and is similar to the tweak operator used in our earlier work [28]. (2) *The external loop* (Lines 5 to 16). This loop tries to find an optimized solution through random search. That is, the search in the outer loop is *explorative*. More precisely, the algorithm combines an exploitative search with an explorative search. After performing an exploitative search for a given amount of time (i.e., $restartTime$), it restarts the search and moves to a randomly selected point (Line 15) and resumes the exploitative search from the new randomly selected point. The algorithm stops after it reaches a given time budget (Line 15).

We discuss two important points about our test generation algorithm: (1) Each candidate solution in our search algorithm is a test suite of size k. This is similar to the approach taken in the *whole test suite generation* algorithm proposed by Fraser and Arcuri in [22]. The reason we use a whole test suite generation algorithm instead of generating test cases individually is that computing fitnesses for one test case and for several test cases takes almost the same amount of time. This is because, in our work, the most time consuming operation is to load a Simulink model. Once the model is loaded, the time required to run several test cases versus

one test case is not very different. Hence, we decided to generate and mutate the k test cases at the same time. (2) Our algorithm does not require test oracles to generate new test cases. Note that computing fit_{Dis} and fit_{dbb} only requires test execution slices without any pass/fail information. To compute fit_{Dens}, in addition to test execution slices, we need static backward slices that can be obtained from Simulink models. Test oracle information for the k new test cases is only needed after test generation in subroutine STATISTICALDEBUGGING (see Figure 2) when a new statistical ranking is computed. In the next section, we discuss the STOPTESTGENERATION subroutine (see Figure 2) that allows us to stop test generation before performing all the test generation rounds when we can predict situations where test generation is unlikely to improve the fault localization.

B. Stopping Test Generation

As noted in the literature [15], adding test cases does not always improve statistical debugging results. Given that in our context test oracles are expensive, we provide a strategy to stop test generation when adding new test cases is unlikely to bring about noticeable improvements in the fault localization results. Our STOPTESTGENERATION subroutine is shown in Figure 4. It has two main parts: In the first part (Lines 1–6), it tries to determine if the decision about stopping test generation can be made only based on the characteristics of $newList$ (i.e., the latest generated ranked list) and static analysis of Simulink models. For this purpose, it computes Simulink super blocks and compares the top ranked groups of $newList$ with Simulink super blocks. In the second part (Lines 7-10), our algorithm relies on a predictor model to make a decision about further rounds of test generation. We build the predictor model using supervised learning techniques (i.e., *decision trees* [23]) based on the following three features: (1) the current test generation round, (2) the *SetDistance* between the latest ranked list and the initial ranked list, and (3) the *OrderingDistance* between the latest ranked list and the initial ranked list. Below, we first introduce Simulink super blocks. We will then introduce *SetDistance* and the *OrderingDistance* that are used as input features for our predictor model. After that, we describe how we build and use our decision tree predictor model.

Super blocks. Given a Simulink model $M = (Nodes, Links, Inputs, Outputs)$, we define a *super block* as the largest set $B \subseteq Nodes$ of (atomic) Simulink blocks such that for every test case tc and every output $o \in Outputs$, we have either $B \subseteq tes_{tc,o}$ or $B \cap tes_{tc,o} = \emptyset$. The definition of super block is very similar to the definition of dynamic basic blocks (DBB) discussed in Section III-A. The only difference is that dynamic basic blocks are defined with respect to the test execution slices generated by a given test suite, while super blocks are defined with respect to test execution slices that can be generated by any potential test case. Hence, dynamic basic blocks can be computed *dynamically* based on test execution slices obtained by the current test suite, whereas super blocks are computed by *static analysis* of the structure of Simulink models. In order

STOPTESTGENERATION()
Input: - r: The index of the latest test generation round
- M: The underlying Simulink model
- $initialList$: A ranked list obtained using an initial test suite
- $newList$: A ranked list obtained at round r after some test cases are added to the initial test suite

Output: $result$: Test generation should be stopped if $result$ is true

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

Fig. 4: The STOPTESTGENERATION subroutine used in our approach (see Figure 2).

to compute super blocks, we identify conditional (control) blocks in the given Simulink model. Each conditional block has an incoming control link and a number of incoming data links. Corresponding to each conditional block, we create some branches by matching each incoming data link with the conditional branch link. We then remove the conditional block and replace it with the new branches. This allows us to obtain a behaviorally equivalent Simulink model with no conditional blocks. We further remove parallel branches by replacing them with their equivalent sequential linearizations. We then use the resulting Simulink model to partition the set $Nodes$ into a number of *disjoint* super blocks $B_1, \ldots, B_l$.

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

Features for building our predictor model. We describe the three features used in our predictor models. The first feature is the test generation round. As shown in Figure 2, we generate test cases in a number of consecutive rounds. Intuitively, adding test cases at the earlier rounds is likely to improve statistical debugging more compared to the later rounds. Our second and third features (i.e., *SetDistance* and *OrderingDistance*) are similarity metrics comparing the latest generated rankings (at the current round) and the initial rankings. These two metrics are formally defined below.

Let *initialList* be the ranking generated using an initial test suite, and let *newList* be the latest generated ranking. Let $rg_1^{new}, \ldots, rg_m^{new}$ be the ranked groups in *newList*, and $rg_1^{initial}, \ldots, rg_{m'}^{initial}$ be the ranked groups in *initialList*. Our *SetDistance* feature computes the dissimilarity between the top-N ranked groups of *initialList* and *newList* using the *intersection metric* [29]. We focus on comparing the top N

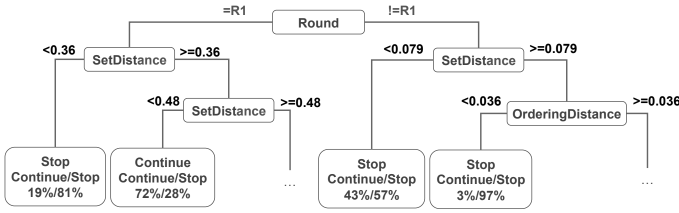

Fig. 5: A snapshot example of a decision tree.

ranked groups because, in practice, the top ranked groups are primarily inspected by engineers. We compute the *SetDistance* based on the average of the overlap between the top-N ranked groups of the two ranked lists. Formally, we define the *SetDistance* between $initialList$ and $newList$ as follows.

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

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

Our third feature is *OrderingDistance*. Similar to *SetDistance*, the *OrderingDistance* feature also attempts to compute the dissimilarity between the top-N ranked groups of $initialList$ and $newList$. However, in contrast to *SetDistance*, *OrderingDistance* focuses on identifying changes in pairwise orderings of blocks in the rankings. In particular, we define *OrderingDistance* based on *Kendall Tau Distance* [30] that is a well-known measure for such comparisons. This measure computes the dissimilarity between two rankings by counting *the number of discordant pairs* between the rankings. A pair b and b' is discordant if b is ranked higher than b' in $newList$ (respectively, in $initialList$), but not in $initialList$ (respectively, in $newList$). In our work, in order to define the *OrderingDistance* metric, we first create two sets $initialL$ and $newL$ based on $initialList$ and $newList$, respectively: $initialL$ is the same as $initialList$ except that all the blocks that do not appear in the top-N ranked groups of neither $initialList$ nor $newList$ are removed. Similarly, $newL$ is the same as $newList$ except that all the blocks that do not appear in the top-N ranked groups of neither $newList$ nor $initialList$ are removed. Note that $newL$ and $initialL$ have the same number blocks. We then define the *OrderingDistance* metric as follows:

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

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

Prediction model. Our prediction model builds on an intuition that by comparing statistical rankings obtained at the current and previous rounds of test generation, we may be able to predict whether further rounds of test generation are useful or not. We build a prediction model based on the three features discussed above (i.e., the current round, *SetDistance*, *OrderingDistance*). We use supervised learning methods, and in particular, decision trees [23]. The prediction model returns a binary answer indicating whether the test generation should stop or not. To build the prediction model, we use historical data consisting of statistical rankings obtained during a number of test generation rounds and *fault localization accuracy* results corresponding to the statistical rankings. When such historical data is not available the prediction model always recommends that test generation should be continued. After applying our approach (Figure 2) for a number of rounds, we gradually obtain the data that allows us to build a more effective prediction model that can recommend to stop test generation as well. Specifically, suppose $rankList$ is a ranking obtained at round r of our approach (Figure 2), and suppose $initList$ is a ranking obtained initially before generating test cases (Figure 2). The accuracy of fault localization for $rankList$ is *the maximum number of blocks inspected* to find a fault when engineers use $rankList$ for inspection. To build our decision tree, for each $rankList$ computed by our approach in Figure 2, we obtain the triple $I = (r, SetDistance(initList, rankList), OrderingDistance(initList, rankList))$. We then compute the maximum fault localization accuracy improvement that we can achieve if we proceed with test generation from round r (the current round) until the last round of our algorithm in Figure 2. We denote the maximum fault localization accuracy improvement by $Max_ACC_r(rankList)$. We then label the triple I with `Continue`, indicating that test generation should continue, if $Max_ACC_r(rankList)$ is more than a threshold (THR); and with `Stop`, indicating that test generation should stop, if $Max_ACC_r(rankList)$ is less than the threshold (THR). Note that THR indicates the minimum accuracy improvements that engineers expect to obtain to be willing to undergo the overhead of generating new test cases.

Having obtained triples I labelled with `Stop` or `Continue`, we build our decision tree model (prediction model). Decision trees are composed of leaf nodes, which represent *partitions*, and non-leaf nodes, which represent *decision variables*. A decision tree model is built by partitioning the set of input triples in a stepwise manner aiming to create partitions with increasingly more homogeneous labels (i.e., partitions in which the majority of triples are labelled either by `Stop` or by `Continue`). The larger the difference between the number of triples with `Stop` and `Continue` in a partition, the more homogeneous that partition is. Decision variables (i.e., non-leaf node) in our decision tree model represent logical conditions on the input features (i.e., r, *SetDistance*, or *OrderingDistance*). Figure 5 shows a fragment of our decision tree model. For example, the model shows, among the triples satisfying r = `R1` and $SetDistance < 0.36$ conditions, 81% are labelled with `Stop` and 19% are labelled with `Continue`.

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

Stop Test Generation Algorithm. The STOPTESTGENERATION() algorithm starts by identifying the super blocks in

$newList$, the latest generated ranking (Line 2). If it happens that the top-N ranked groups in $newList$ all comprise a single super block, then test generation stops (Line 3-4), because such ranking cannot be further refined by test generation. If we are in the first round (i.e., $r = 0$), the algorithm returns *false*, meaning that test generation should continue. For all other rounds, we use the decision tree prediction model. Specifically, we compute the *SetDistance* and *OrderingDistance* features corresponding to $newList$, and pass these two values as well as r (i.e., the round) to the prediction model. The prediction model returns *true*, indicating that test generation should be stopped, if the three input features satisfy a sequence of conditions leading to a (leaf) partition where at least 95% of the elements in that partition are labelled `Stop`. Otherwise, our prediction model returns *false*, indicating that test generation should be continued. For example, assuming the decision tree in Figure 5 is our prediction model, we stop test generation only if we are not in round one, *SetDistance* is greater than or equal to 0.079, and *OrderingDistance* is less than 0.036. This is because, in Figure 5, these conditions lead to the leaf partition with 97% stop-labelled elements.

IV. Empirical Evaluation

A. Research Questions

RQ1. [Evaluating and comparing different test generation fitness heuristics] *How is the fault localization accuracy impacted when we apply our search-based test generation algorithm in Figure 3 with our three selected fitness functions (i.e., coverage dissimilarity (f_{Dis}), coverage density (f_{Dens}), and number of dynamic basic blocks (f_{dbb}))?* We report the fault localization accuracy of a ranking generated by an initial test suite compared to that of a ranking generated by a test suite extended using our algorithm in Figure 3 with a small number of test cases. We further compare the fault localization accuracy improvement when we use our three alternative fitness functions, and when we use a random test generation strategy not guided by any of these fitness functions.

RQ2. [Evaluating impact of adding test cases] *How does the fault localization accuracy change when we apply our search-based test generation algorithm in Figure 3?* We note that adding test cases does not always improve the fault localization accuracy [15]. With this question, we investigate how often fault localization accuracy improves after adding test cases. In particular, we apply our approach in Figure 2 without calling the StopTestGeneration subroutine, and identify how often subsequent rounds of test generation do not lead to fault localization accuracy improvement.

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

B. Experiment Settings

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

Industrial Subjects. In our experiment, we use three Simulink models referred to as *MA*, *MZ* and *MGL*, and developed by Delphi Automotive [31]. Table II 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 [32].

TABLE II: Key information about industrial subjects.

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

We asked a Delphi engineer to seed 20 realistic and typical faults into each model. We have provided detailed descriptions of the seeded faults in [33]. In total, we generated 60 faulty versions (one fault per each faulty version). We ensured that the faults were of different types and were seeded into different parts of the models. All experiment data and scripts are available in [33].

Test Suite and Test Oracles. We generated three initial test suites (i.e., TS in Figure 2) for MA, MZ 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. Given that in our work we assume test oracles are manual, we aim to generate test suites that are not large. However, the test suites should be large enough to generate a meaningful statistical ranking. Hence, at least some test cases in the test suite exhibit failures. In our work, we chose to use initial test suites with size 10. To enable the full automation of our experiments, we used the fault-free versions of our industrial subjects as test oracles. On average, our initial test suites covered 75.5% of the structure of the faulty models.

Experiment Design. To answer our research questions, we applied our approach to the faulty versions of our three models, in total 60 faulty versions. We refer to the test generation algorithm in Figure 3 as HCRR since it builds on the HCRR search algorithm. We refer to HCRR when it is used with fitness functions f_{Dis}, f_{Dens} and f_{dbb} as HCRR-Dissimilarity, HCRR-Density and HCRR-DBB, respectively. We set both the number of new test cases per round (i.e., k in Figure 2), and the number of rounds (i.e., $round$ in Figure 2) to five. That is, in total, we generate 25 new test cases by applying our approach. We applied our three alternative HCRR algorithms to our 60 faulty versions. We ran each HCRR algorithm for 45 minutes with two restarts. To account for randomness of the search algorithms, we repeat our experiments for ten times (i.e., ten trials). Further, to compute input features for our stopping criteria setting, we set N (in Figure 4) to five. We ran our

experiment on a high performance computing platform [35] with 2 clusters, 280 nodes, and 3904 cores. Our experiment were executed on different nodes of a cluster with Intel Xeon L5640@2.26GHz processor. In total, our experiment (using a single node 4 cores) required 6750 hours. Most of the experiment time was used to execute the generated test cases in Simulink. In total, we generated and executed 129000, 159000, and 120000 test cases for *MA MZ*, and *MGL*, respectively.

C. Evaluation Metrics

We evaluate the accuracy of the rankings generated at different rounds of our approach using the following metrics [8], [9], [24], [36]–[38]: the *absolute number of blocks inspected* to find faults, and the *proportion of faults localized* when engineers inspect fixed numbers of the top most suspicious blocks. The former was already discussed for prediction models in Section III-B. The proportion of faults localized is the proportion of localized faults over the total number of faults when engineers inspect a fixed number of the top most suspicious blocks from a ranking.

D. Experiment Results

RQ1. [Evaluating and comparing different test generation fitness heuristics] Figure 6 compares the fault localization results after applying HCRR-DBB, HCRR-Density and HCRR-Dissimilarity algorithms to generate 25 test cases (five test cases in five rounds) with the fault localization results obtained before applying these algorithms (i.e., Initial) and with the fault localization results obtained after generating 25 test cases randomly (i.e., Random). In particular, in Figure 6(a), we compare the distributions of the maximum number of blocks inspected to locate faults (i.e. accuracy) in our 60 faulty versions when statistical rankings are generated based on the initial test suite (i.e. Initial), or after using HCRR-DBB, HCRR-Density, HCRR-Dissimilarity and Random test generation to add 25 test cases to the initial test suite. Each point in Figure 6(a) represents fault localization accuracy for one run of one faulty version. According to Figure 6(a), before applying our approach (i.e., Initial), engineers on average need to inspect at most 76 blocks to locate faults. When in addition to the initial test suite, we use 25 *randomly* generated test cases, the maximum number of blocks inspected decreases to, on average, 62 blocks. Finally, engineers need to inspect, on average, 42.4, 44 and 42.8 blocks if they use the rankings generated by HCRR-DBB, HCRR-Density and HCRR-Dissimilarity, respectively. We performed non-parametric pairwise Wilcoxon signed-rank tests to check whether the improvement on the number of blocks inspected is statistically significant. The results show that the fault localization accuracy distributions obtained by HCRR-DBB, HCRR-Density and HCRR-Dissimilarity are significantly lower (better) than those obtained by Random and Initial (with p-value<0.0001).

Similarly, Figure 6(b) shows the proportion of faults localized when engineers inspect a fixed number of blocks in the rankings generated by Initial, and after generating 25 test

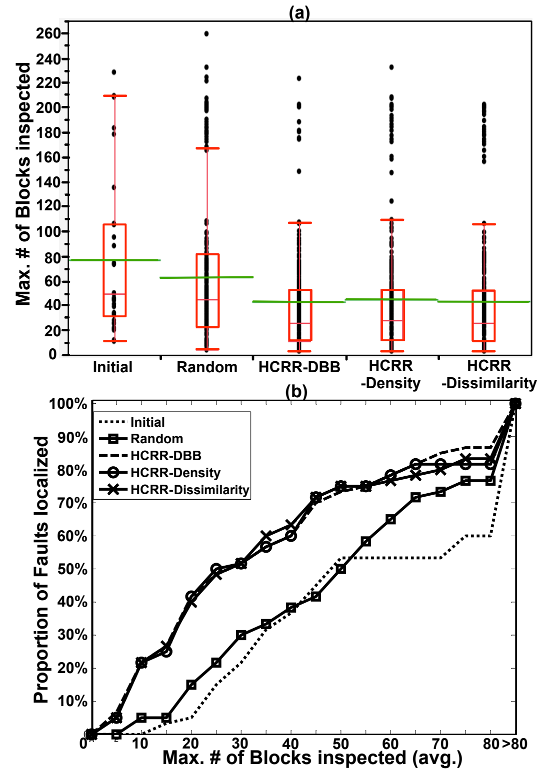

Fig. 6: Comparing the number of blocks inspected (a) and the proportion of faults localized (b) before and after applying HCRR-DBB, HCRR-Dissimilarity and HCRR-Density, and with Random test generation (i.e., Random).

cases with HCRR-DBB, HCRR-Density, HCRR-Dissimilarity, and Random. Specifically, the X-axis shows the number of top ranked blocks (ranging from 10 to 80), and the Y-axis shows the proportion of faults among a fixed number of top ranked blocks in the generated rankings. Note that, in Figure 6(b), the maximum number of blocks inspected (X-axis) is computed as an average over ten trials for each faulty version. According to Figure 6(b), engineers can locate faults in 13 out of 60 (21.67%) faulty versions when they inspect at most 10 blocks in the rankings generated by any of our techniques i.e., HCRR-DBB, HCRR-Density and HCRR-Dissimilarity. However, when test cases are generated randomly, by inspecting the top 10 blocks, engineers can locate faults in only 3 out of 60 (5%) faulty versions. As for the rankings generated by the initial test suite, no faults can be localized by inspecting the top 10 blocks. Using HCRR-DBB, HCRR-Density and HCRR-Dissimilarity, on average, engineers can locate 50% of the faults in the top 25 blocks of each ranking. In contrast, when engineers use the initial test suite or a random test generation strategy, in order to find 50% of the faults, they need to inspect, on average, 50 blocks in each ranking.

In summary, the test cases generated by our approach are able to help significantly improve the accuracy of fault localization results. In particular, by adding a small number of test cases (i.e., only 25 test cases), we are able to reduce the average number of blocks that engineers need to inspect to find

a fault from 76 to 43 blocks (i.e., 43.4% reduction). Further, we have shown that the fault localization accuracy results obtained based on HCRR-DBB, HCRR-Density and HCRR-Dissimilarity are significantly better than those obtained by a random test generation strategy. Specifically, with Random test generation, engineers need to inspect an average of 62 blocks versus an average of 43 blocks when HCRR-DBB, HCRR-Density and HCRR-Dissimilarity are used.

RQ2. [Evaluating impact of adding test cases] We evaluate the fault localization accuracy of the ranking results obtained at each test generation round. In particular, we computed the fault localization accuracy of rankings obtained by applying HCRR-DBB, HCRR-Density and HCRR-Dissimilarity to our 60 faulty versions from round one to five where at each round five new test cases are generated. Recall that we have repeated 10 times each application of our technique to each faulty model. That is, in total, we have 1800 trials (60 faulty versions $\times$ 3 algorithms $\times$ 10 runs). Among these 1800 trials, we observed that, as we go from round one to round five, in 953 cases (i.e., 53%), the fault localization accuracy improves at every round; in 803 cases (i.e., 44.6%), the accuracy improves at some (but not all) rounds; and in 44 cases (i.e., 2.4%), the accuracy never improves at any of the rounds from one to five.

To explain why adding new test cases does not always improve fault localization accuracy, we investigate the notion of Coincidentally Correct Test cases (CCT) for Simulink [13]. CCTs are test execution slices that execute faulty blocks but do not result in failure. We note that as we add new test cases, the number of CCTs may either stay the same or increase. In the former case, the fault localization accuracy either stays the same or improves. However, in the latter case, the accuracy changes will be unpredictable.

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

RQ3. [Effectiveness of our StopTestGeneration subroutine] In order to generate the prediction model used in the StopTestGeneration subroutine, we consider all the statistical ranking results obtained by applying the five rounds of test generation to the 60 faulty versions as well as the corresponding accuracy results. We randomly divide the results into three sets, and use one of these sets to build the decision tree prediction model (i.e., as a training set). The other two sets are used to evaluate the decision tree prediction model (i.e., as test sets). Following a standard cross-validation procedure, we follow this process three times so that each set is used as the training set at least once. To build these models, we set $THR = 15$ (i.e., the threshold used to determine the `Stop` and the `Continue` labels in Section III-B). That is, engineers are willing to undergo the overhead of adding new test cases if the fault localization accuracy is likely to improve by at least 15 blocks. Figure 7(a) shows the fault localization accuracy results (i.e., the maximum number of blocks inspected)

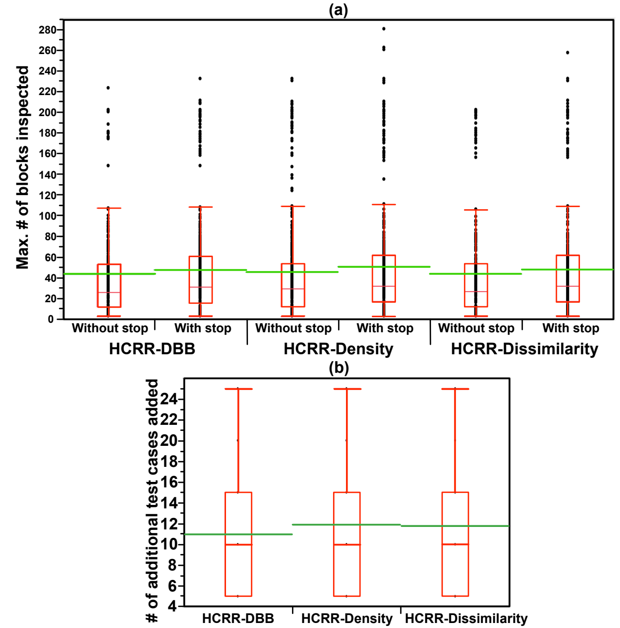

Fig. 7: The maximum number of blocks inspected and the number of new test cases added when we applied StopTestGeneration on the rankings generated using HCRR-DBB, HCRR-Density, and HCRR-Dissimilarity based on the predictor models obtained for three different validation sets.

obtained by our three test generation algorithms (HCRR-DBB, HCRR-Density, and HCRR-Dissimilarity) and when the StopTestGeneration subroutine is used with the three decision tree prediction models generated by cross-validation. These results are shown in columns with `With stop` label. Figure 7(a), further, shows the accuracy results obtained by applying the five rounds *without* using StopTestGeneration in columns labelled `Without stop`. In addition, Figure 7(b) shows the number of new test cases generated by HCRR-DBB, HCRR-Density and HCRR-Dissimilarity when we applied the StopTestGeneration subroutine. Note that we generate 25 test cases in five rounds without StopTestGeneration.

According to Figure 7, we are able to obtain almost the same fault localization accuracy with considerably fewer number of new test cases when we use the StopTestGeneration subroutine compared to when we do not use it. In particular, on average, when we use the StopTestGeneration subroutine, the fault localization accuracies obtained for HCRR-DBB, HCRR-Dissimilarity and HCRR-Density are 47.3, 47.9 and 50.4, respectively. In contrast, without the StopTestGeneration subroutine, the fault localization accuracies obtained for HCRR-DBB, HCRR-Dissimilarity and HCRR-Density are 43, 43.4 and 45.1, respectively. We note that these accuracies are obtained by only generating, on average, 11 test cases for HCRR-DBB, and 12 test cases for both HCRR-Density and HCRR-Dissimilarity. We have also repeated our experiments for $THR = 10$. The results for $THR = 10$ show that the average fault localization accuracies improve by one to two blocks while the number of new test cases also increases by one or two when compared with the results for $THR = 15$.

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

V. RELATED WORK

Many test generation techniques have been proposed for different purposes e.g., maximizing program coverage ([22], [39]–[52]) and revealing faults ([43], [44], [53]–[64]) for programs, or maximizing structural coverage [65]–[74] and revealing faults [19], [28], [75]–[83] for Simulink models. Nevertheless, only a few test generation techniques aim to improve fault localization accuracy. These techniques specifically focus on Java/C programs [15]–[17] and on web applications [18]. Our work aims to improve fault localization accuracy for Simulink models by extending an existing test suite with a *small* number of test cases. This is to ensure applicability of our approach in situations where test oracles are developed manually or running test cases is expensive.

One important requirement in our work is that the pass/-fail information for each candidate test input is not readily available, and hence, test generation techniques that require such information to improve fault localization [17], [18] are not applicable in our case since these techniques are feasible only when test oracles are automatable. Hence, in our work, we identify the test generation techniques of [16] and [15] that satisfy our requirement. Both of these techniques attempt to generate test cases that execute varying subsets of program statements. In particular, Baudry et. al. [16] guide test generation by maximizing the number of Dynamic Basic Blocks (i.e. program elements that are always executed together), and Campos et. al. [15] attempt to generate test cases with a balanced number of long and short structural test coverages. In our work, we adapt these two test generation algorithms to Simulink models. In addition, we introduce a new test generation objective that has previously been used for test prioritization [21] and use it to improve fault localization for Simulink models. In contrast to the work of [15], [16], [21], we assess the capabilities of test generation techniques in improving Simulink fault localization when the number of newly generated test cases is small. We, further, combine these techniques with a predictor model that stops test generation when new test cases are not likely to help improve fault localization accuracy.

Le and Lo [84] propose an approach to predict fault localization accuracy based on features extracted from statistical rankings generated by a fixed and specific test suite. Our predictor model instead is built based on features that compare statistical rankings generated by a test suite and its extensions. Moreover, our predictor model is used to help stop test generation and to ensure test suite minimality. Further investigation is required to assess the effectiveness of the features proposed in [84] as a test generation stopping criterion.

Xia et al. [85] select a subset of a given test suite such that the fault localization accuracy achieved by the subset is the same as the accuracy achieved by the entire test suite. Similar to our work, they create predictor models based on changes in rankings as new test cases are added to the underlying test suite. However, they build a predictor model for each program element as opposed to our work where we build one predictor model based on the changes in the top-N ranked groups. As discussed earlier, since Simulink atomic blocks in the same super block always have the same rank, creating separate predictors for each individual atomic blocks is too fine-grained and redundant. Furthermore, at each round, in order to select a test case, Xia et al. [85] need to compare the spectra of the candidate test case with those of all the remaining test cases. This makes their approach computationally and memory intensive when the test suite from which test cases are selected is large. In our work, however, we extend an initial test suite using a search-based test generation technique guided by objectives that aim to increase test suite diversity without any need to compare the spectra of many test cases.

VI. CONCLUSION

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

In future, we intend to study fault localization for evolving Simulink models. A recent study of industrial Simulink models indicates a strong co-evolution relation between changes in models and in their corresponding test suites [86]. We plan to investigate how such relations can be used to generate test suites that lead to effective Simulink fault localization, especially, when models are subject to frequent changes.

ACKNOWLEDGMENTS

This project has received funding from Luxembourg's National Research Fund (grant agreement numbers FNR/P10/03 and FNR-8003491), Delphi Automotive Systems and the European Research Council under the European Union's Horizon 2020 research and innovation program (grant agreement number 694277).

REFERENCES

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

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

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

[5] 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*. IEEE, 2007, pp. 89–98.

[6] 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*, 2002, pp. 467–477.

[7] 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.

[8] 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.

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

[10] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, "Lightweight fault-localization using multiple coverage types," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE, 2009, pp. 56–66.

[11] E. Wong, T. Wei, Y. Qi, and L. Zhao, "A crosstab-based statistical method for effective fault localization," in *Proceedings of the International Conference on Software Testing, Verification, and Validation*, 2008, pp. 42–51.

[12] E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2014.

[13] B. Liu, Lucia, S. Nejati, L. Briand, and T. Bruckmann, "Simulink fault localization : an iterative statistical debugging approach," *Software Testing, Verification and Reliability*, pp. 431–459, 2016.

[14] ——, "Localizing multiple faults in simulink models," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2016, pp. 146–156.

[15] J. Campos, R. Abreu, G. Fraser, and M. d'Amorim, "Entropy-based test generation for improved fault localization," in *the IEEE/ACM 28th International Conference on Automated Software Engineering*. IEEE, 2013, pp. 257–267.

[16] B. Baudry, F. Fleurey, and Y. Le Traon, "Improving test suites for efficient fault localization," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 82–91.

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

[18] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Directed test generation for effective fault localization," in *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 2010, pp. 49–60.

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

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

[21] B. Jiang, Z. Zhang, W. K. Chan, and T. Tse, "Adaptive random test case prioritization," in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 233–244.

[22] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.

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

[24] 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*, 2005, pp. 273–282.

[25] S. Luke, "Essentials of metaheuristics," vol. 113, 2009.

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

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

[28] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann, "Effective test suites for mixed discrete-continuous stateflow controllers," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 84–95.

[29] R. Fagin, R. Kumar, and D. Sivakumar, "Comparing top k lists," *SIAM Journal on Discrete Mathematics*, vol. 17, no. 1, pp. 134–160, 2003.

[30] M. G. Kendall, "A new measure of rank correlation," *Biometrika*, vol. 30, pp. 81–93, 1938.

[31] Delphi Automotive, "Delphi automotive luxembourg," http://www.delphi.com/about/locations/luxembourg.

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

[33] B. Liu, "experiment related." https://github.com/Avartar/TCGenForFL/.

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

[35] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos, "Management of an academic hpc cluster: The ul experience," in *Proc. of the Intl. Conf. on High Performance Computing & Simulation*. Bologna, Italy: IEEE, July 2014, pp. 959–967.

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

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

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

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

[40] N. Williams, B. Marre, P. Mouy, and M. Roger, "Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis," in *European Dependable Computing Conference*. Springer, 2005, pp. 281–292.

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

[42] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 263–272.

[43] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *ACM Sigplan Notices*, vol. 40, no. 6. ACM, 2005, pp. 213–223.

[44] N. Tillmann and J. De Halleux, "Pex–white box test generation for. net," in *International conference on tests and proofs*. Springer, 2008, pp. 134–153.

[45] B. Korel, "Automated software test data generation," *IEEE Transactions on software engineering*, vol. 16, no. 8, pp. 870–879, 1990.

[46] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information and Software Technology*, vol. 43, no. 14, pp. 841–854, 2001.

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

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

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

[50] P. Tonella, “Evolutionary testing of classes,” in *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4. ACM, 2004, pp. 119–128.

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

[52] S. Wappler and F. Lammermann, “Using evolutionary algorithms for the unit testing of object-oriented software,” in *Proceedings of the 7th annual conference on Genetic and evolutionary computation*. ACM, 2005, pp. 1053–1060.

[53] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, “Finding bugs in dynamic web applications,” in *Proceedings of the international symposium on Software testing and analysis*. ACM, 2008, pp. 261–272.

[54] G. Fraser and A. Zeller, “Mutation-driven generation of unit tests and oracles,” *IEEE Transactions on Software Engineering*, vol. 38, no. 2, pp. 278–292, 2012.

[55] R. DeMilli and A. J. Offutt, “Constraint-based automatic test data generation,” *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900–910, 1991.

[56] A. J. Offutt, Z. Jin, and J. Pan, “The dynamic domain reduction procedure for test data generation,” *Software-Practice and Experience*, vol. 29, no. 2, pp. 167–93, 1999.

[57] B. F. Jones, D. E. Eyres, and H.-H. Sthamer, “A strategy for using genetic algorithms to automate branch and fault-based testing,” *The Computer Journal*, vol. 41, no. 2, pp. 98–107, 1998.

[58] K. Ayari, S. Bouktif, and G. Antoniol, “Automatic mutation test input data generation via ant colony,” in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. ACM, 2007, pp. 1074–1081.

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

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

[61] C. Pacheco and M. D. Ernst, “Eclat: Automatic generation and classification of test inputs,” in *European Conference on Object-Oriented Programming*. Springer, 2005, pp. 504–527.

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

[63] M. A. Alipour, A. Groce, R. Gopinath, and A. Christi, “Generating focused random tests using directed swarm testing,” in *Proceedings of the international symposium on Software testing and analysis*. ACM, 2016.

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

[65] A. Windisch, “Search-based testing of complex simulink models containing stateflow diagrams,” in *Proceeding of the 31st International Conference on Software Engineering-Companion*. IEEE, 2009, pp. 395–398.

[66] ——, “Search-based test data generation from stateflow statecharts,” in *Proceedings of the 12th annual conference on Genetic and evolutionary computation*. ACM, 2010, pp. 1349–1356.

[67] S. Mohalik, A. A. Gadkari, A. Yeolekar, K. Shashidhar, and S. Ramesh, “Automatic test case generation from simulink/stateflow models using model checking,” *Software Testing, Verification and Reliability*, vol. 24, no. 2, pp. 155–180, 2014.

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

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

[70] S. Sims and D. C. DuVarney, “Experience report: the reactis validation tool,” in *ACM SIGPLAN Notices*, vol. 42, no. 9. ACM, 2007, pp. 137–140.

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

[72] A. A. Gadkari, A. Yeolekar, J. Suresh, S. Ramesh, S. Mohalik, and K. Shashidhar, “Automotgen: Automatic model oriented test generator for embedded control systems,” in *International Conference on Computer Aided Verification*. Springer, 2008, pp. 204–208.

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

[74] M. Satpathy, A. Yeolekar, P. Peranandam, and S. Ramesh, “Efficient coverage of parallel and hierarchical stateflow models for test case generation,” *Software Testing, Verification and Reliability*, vol. 22, no. 7, pp. 457–479, 2012.

[75] Y. Zhan and J. A. Clark, “Search-based mutation testing for simulink models,” in *Proceedings of the 7th annual conference on Genetic and evolutionary computation*. ACM, 2005, pp. 1061–1068.

[76] ——, “A search-based framework for automatic testing of matlab/simulink models,” *Journal of Systems and Software*, vol. 81, no. 2, pp. 262–285, 2008.

[77] A. Brillout, N. He, M. Mazzucchi, D. Kroening, M. Purandare, P. Rümmer, and G. Weissenbacher, “Mutation-based test case generation for simulink models,” in *International Symposium on Formal Methods for Components and Objects*. Springer, 2009, pp. 208–227.

[78] R. Cleaveland, S. A. Smolka, and S. T. Sims, “An instrumentation-based approach to controller model validation,” in *Automotive Software Workshop*. Springer, 2006, pp. 84–97.

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

[80] M. Mazzolini, A. Brusaferri, and E. Carpanzano, “Model-checking based verification approach for advanced industrial automation solutions,” in *Proceeding of the IEEE Conference on Emerging Technologies and Factory Automation*. IEEE, 2010, pp. 1–8.

[81] R. Venkatesh, U. Shrotri, P. Darke, and P. Bokil, “Test generation for large automotive models,” in *Proceeding of the IEEE International Conference on Industrial Technology*. IEEE, 2012, pp. 662–667.

[82] D. Holling, A. Pretschner, and M. Gemmar, “8cage: lightweight fault-based test generation for simulink,” in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 859–862.

[83] D. Balasubramanian, C. S. Păsăreanu, M. W. Whalen, G. Karsai, and M. Lowry, “Polyglot: modeling and analysis for multiple statechart formalisms,” in *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 45–55.

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

[85] X. Xia, L. Gong, T.-D. B. Le, D. Lo, L. Jiang, and H. Zhang, “Diversity maximization speedup for localizing faults in single-fault and multi-fault programs,” *Automated Software Engineering*, vol. 23, no. 1, pp. 43–75, 2016.

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