**Noname manuscript No.**
(will be inserted by the editor)

# Effective Fault Localization of Automotive Simulink Models: Achieving the Trade-Off between Test Oracle Effort and Fault Localization Accuracy

**Bing Liu · Shiva Nejati · Lucia · Lionel C. Briand**

Received: date / Accepted: date

**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 four test objectives that aim to increase test suite diversity. We use four 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) expanding test suites used for fault localization using any of our four test objectives, even when the expansion is small, can significantly improve the accuracy of fault localization, (2) varying test objectives used to generate the initial test suites for fault localization does not have a significant impact on the fault localization results obtained based on those test suites, and (3) we identify an optimal configuration for prediction models to help stop test generation when it is unlikely to be beneficial. We further show that our optimal 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.

**Keywords** Fault localization · Simulink models · search-based testing · test suite diversity · supervised learning

## 1 Introduction

The embedded software industry increasingly relies on model-based development methods to develop software components [69]. These components are largely de-

B. Liu, S. Nejati, Lucia, L. Briand
SnT Centre, University of Luxembourg, Luxembourg
E-mail: liu@svv.lu, nejati@svv.lu, lucia.lucia@uni.lu, briand@svv.lu

veloped in the Matlab/Simulink language [50]. 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) [15, 79]. 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 (also known as statistical fault localization) is a lightweight and well-studied debugging technique [1, 34, 39, 44, 62, 64, 75, 76]. 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 [42], 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 [42]. We further extended our approach to handle fault localization for Simulink models with multiple faults [41].

Since statistical debugging is essentially heuristic, despite various research advancements, it still remains largely unpredictable [17]. In practice, it is likely that several elements have the same suspiciousness score as that of the faulty one, 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 [8, 11, 17, 63].

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 most domains (e.g., automotive), it is expensive to execute Simulink models and to characterize their expected behaviour [54, 79]. This is because Simulink models include computationally expensive physical models [12], and their outputs are complex continuous signals [54]. We identify four alternative test objectives that aim to generate test cases exercising diverse parts of the underlying code and adapt these objectives to Simulink models [5, 11, 17, 32]. We use these objectives to develop a search-based test generation algorithm, which builds on the whole test suite generation algorithm [23], to extend an existing test suite with a small number of test cases. In our work, we opt for single-state search algorithms as op-

posed to population-based ones. This is because population-based search algorithms have to execute a set of candidate test cases (a population) at each iteration [46]. Hence, they are less likely to scale when running test cases is expensive.

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 beneficial: 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 any test case $tc$ executes either *all* or *none* of the blocks in $B$. That is, there is no test case that executes a subset (and not all) of the blocks in a super block. Statistical debugging, by definition, always ranks the blocks inside a super block together in the same rank group. Thus, when elements in a rank group are all from a super block, the rank group cannot be further refined through statistical debugging, and hence, test generation is not beneficial. Second, we develop a prediction model based on supervised learning techniques, specifically decision trees [14] 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 [5, 11, 17, 32] 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 four alternative test objectives are almost 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 significantly higher than the improvement obtained by random test generation (baseline). **(ii)** Varying test generation techniques used to build a (small) initial test suite for fault localization does not have a significant impact on the fault localization results. In particular, we experimented with three test generation techniques (i.e., adaptive random, coverage-based and output diversity) to build test suites to generate initial statistical rankings based on which the initial test suites can be expanded. Our experiments show that the fault localization accuracy results corresponding to these three techniques were almost identical. **(iii)** Our strategy based on static analysis and supervised learning is able to stop generating test cases that are not beneficial for fault localization. Specifically, we configured an optimal prediction model by experimenting with different input features for supervised learning and different threshold values for labeling the training data. Our results show that using our optimal prediction model, 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.

This paper extends a previous conference paper [43] published at the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2017). This paper offers major extensions over our previous paper in the following areas: (1) We consider a new test objective, output diversity [53], and study its effectiveness for fault localization. We further compare output diversity with the three test objectives (i.e., coverage dissimilarity, coverage density and number of dynamic basic blocks) discussed in our previous paper [43]. Our results do not indicate a significant difference in fault localization accuracy results obtained based on output diversity compared to the accuracy results obtained based on the test objectives discussed in our previous work. (2) We evaluate the impact of varying test generation techniques used for building initial test suites on fault localization accuracy. Our experiments do not show any significant differences among fault localization accuracies obtained based on different initial test suites generated using different test generation techniques. (3) To build prediction models, we propose a new feature, *RankCorrelation*, defined over statistical rankings. We consider seven different feature combinations consisting of *RankCorrelation* and the three features (i.e., *Round*, *SetDistance*, *OrderingDistance*) discussed in our previous paper [43]. Among these seven feature combinations, the best performing combination is the one consisting of *Round*, *SetDistance* and *OrderingDistance*. (4) We study the impact of changing the threshold value used to label training data for prediction models on the trade-off between fault localization accuracy and the number of new test cases. Based on our results, we provide recommended threshold values that lead to high fault localization accuracies obtained based on a small number of new test cases, hence requiring low effort to develop new test oracles. (5) We present the tool we have implemented to support different parts of our approach. (6) Finally, compared to our previous paper, we have expanded the related work and threats to validity discussions.

This paper is organized as follows: In Section 2, we provide some background on Simulink models and fix our notation. In Section 3, we present our approach for improving fault localization in Simulink models by generating new test cases. In Section 4, we evaluate our work and discuss our experiment results. In Section 5, we briefly introduce the tool support for our approach. In Section 6, we discuss the main threats to validity of our experiments. We then compare our work with the related work in Section 7. We conclude the paper and discuss our future work in Section 8.

## 2 Background and Notation

In this section, we provide some background and fix our formal notation.

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

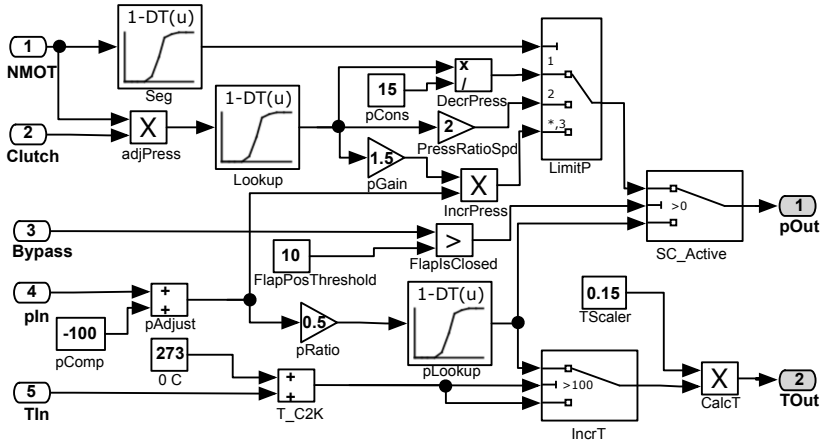


**Fig. 1** A Simulink model example where `pRatio` is faulty.

**Test Input.** Engineers execute (simulate) Simulink models by providing input signals, i.e., functions over time. The simulation length is chosen to be large enough to let the output signals stabilize. In theory, input signals can have any shape. In the automotive domain, however, engineers mostly test Simulink models using constant input signals or step functions (i.e., sequences of input signals). This is because developing test oracles and predicting expected outputs for arbitrary input signals is complex and sometimes infeasible. In this paper, we consider constant input signals only because the subject models used in our evaluation (Section 4) are physical plant models. According to our domain experts, constant input signals are typically sufficient for testing such models. Figure 2(a) shows an input signal example applied to the input `pIn`.

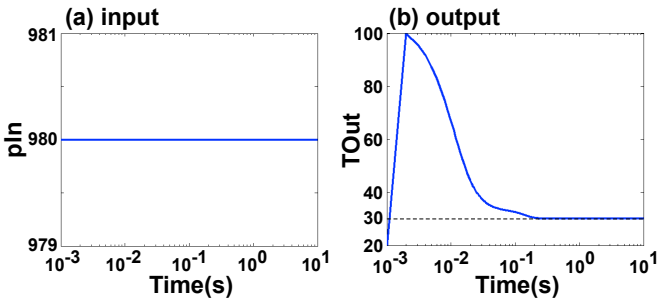


**Fig. 2** Example of an input (a) and an output (b) signal.

**Test Output.** Each test case execution (simulation) of a Simulink model results in an individual output signal for each output port of that model. Engineers evaluate each output signal independently. To determine whether an output passes or fails a test case, engineers check the value at which the output signal stabilizes (if it stabilizes). For example, Figure 2(b) shows an example output signal of `TOut`. As shown in the figure, the output signal stabilizes after 1 sec of simulation. The output values are the

final (stabilized) values of each output signal collected at the end of simulation (e.g., 30 for the signal shown in Figure 2(b)).

**Simulink slicing and Fault Localization.** In our previous work [42], we have shown how statistical debugging can be extended and adapted to Simulink models. Here, we briefly summarize our previous work and present some basic concepts required to defining Statistical debugging for Simulink models. Note that all these concepts have been previously introduced in detail in our previous extended journal article [42]. We further note that, due to a limitation of our slicing approach [42], our fault localization approach is applicable to Simulink models that do not contain any Stateflow (i.e., state machine). This limitation remains valid in our current paper, and hence, we use Simulink models in our evaluation that do not include any Stateflows.

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 [42].

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:

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

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 [42], 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 Fig. 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 1 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 1 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* [33, 34], to rank the Simulink blocks. Note that our comparison [42] of alternative statistical formulas applied to Simulink models revealed no significant difference among these formulas and Tarantula. However, we note that our experiments, both in this paper and in our prior work [42], were performed under the single fault assumption (i.e., none of the faulty models used in our experiments contained more than one fault). If we use models with multiple faults in our experiments, other statistical ranking formulas such as *Ochiai* [1] might perform better than Tarantula. A thorough comparison of the performance of statistical formulas when applied to Simulink models with multiple faults requires further investigations and is left for

**Table 1** Test execution slices and ranking results for Simulink model in Fig. 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 | | |

future work. Finally, we note that although we have used Tarantula in our experiments, our approach is not tied to any specific statistical formula and our experiments are not focusing on comparing such formulas.

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 1 shows the scores and the rank groups for our example in Fig. 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`.

## 3 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 Fig. 3. As the algorithm shows, our approach uses two subroutines TESTGENERATION and STOPTESTGENERATION to improve the standard fault localization based on statistical debugging

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. 3** Overview of our Simulink fault localization approach.

(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 3.1 and 3.2, respectively.

### 3.1 Search-based Test Generation

We use *search-based* techniques [46] 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 four different alternative fitness functions referred to as *coverage dissimilarity*, *coverage density*, *number of dynamic basic blocks* and *output diversity*. Coverage dissimilarity has previously been used for test prioritization [32], and is used in this paper for the first time to improve fault localization. Output diversity has originally been proposed as an alternative to (white-box) structural coverage criteria to generate test suites with high fault revealing ability [5, 53].

The two other alternatives, i.e., *coverage density* [17] and *number of dynamic basic blocks* [11], have been previously used to improve source code fault localization.

*Coverage Dissimilarity.* Coverage dissimilarity aims to increase diversity between test execution slices generated by test cases. We use a set-based distance metric known as *Jaccard distance* [30] to define coverage dissimilarity. Given a pair $tes_{tc,o}$ and $tes_{tc',o'}$ of test execution slices, we denote their dissimilarity as $d(tes_{tc,o}, tes_{tc',o'})$ and define it as follows:

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

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

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

The larger the value of $\mathit{fit}_{Dis}(TS)$, the larger the dissimilarity among test execution slices generated by $TS$. For example, the dissimilarity between test execution slices $tes_{t_1, TOut}$ and $tes_{t_2, TOut}$ in Table 1 is $0.44$. Also, for that example, the average pairwise dissimilarities $\mathit{fit}_{Dis}(TS)$ is $0.71$.

*Coverage Density.* Campos et al [17] argue that the accuracy of statistical fault localization relies on the density of test coverage results. They compute the test coverage density as the average percentage of components covered by test cases over the total number of components in the underlying program. We adapt this computation to Simulink, and compute the coverage density of a test suite $TS$, denoted by $p(TS)$, as follows:

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

That is, our adaptation of coverage density to Simulink computes, for every output $o$, the average size of test execution slices related to $o$ over the static backward slice of $o$. Note that a test execution slice related to output $o$ is always a subset of the static backward slice of $o$. Low values of $p(TS)$ (i.e., close to zero) indicate that test cases cover small parts of the underlying model, and high values (i.e., close to one) indicate that test cases tend to cover most parts of the model. According to Campos et al [17], 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 [17], we define the coverage density fitness function as $\mathit{fit}_{Dens}(TS) = |0.5 - p(TS)|$ and aim to minimize $\mathit{fit}_{Dens}(TS)$ to obtain more accurate ranking results.

*Number of Dynamic Basic Blocks.* Given a test suite $TS$ for fault localization, a *Dynamic Basic Block (DBB)* [11] 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 [11], 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 1 partitions the model blocks in Fig. 1 into six DBBs. An example DBB for that model includes the following blocks: `CalcT`, `TScaler`, `IncrT`, `T_C2K`, `0 C`.

*Output Diversity.* In our previous work [53], we proposed an approach to generate tests for Simulink models based on the notion of *output diversity*. Output diversity is a black-box method that aims to generate a test suite with maximum diversity in its outputs. Our previous work showed that output diversity is effective to reveal faults in Simulink models when test oracles are manual and hence, with test suites of small size [53]. One question that arises here is whether generating test cases based on output diversity can help improve fault localization results. Therefore, we define a test objective to guide our search algorithms based on the notion of output diversity. Simulink models typically contain more than one output (see Section 2). We represent the output of Simulink models as a vector $O = \langle v_1, ..., v_n \rangle$ such that each $v_i$ indicates the value of output $o_i$ of the model under test. We refer to the output vector $O$ generated by a test case $tc$ as $O_{tc}$. Given two output vectors $O_{tc_i} = \langle v_1, ..., v_n \rangle$ and $O_{tc_j} = \langle v_1^{'}, ..., v_n^{'} \rangle$, we define the *normalized* distance between these vectors as follows:

$$dist(O_{tc_i}, O_{tc_j}) = \sum_{j=1}^{n} \frac{|v_j - v_j^{'}|}{max_j - min_j} \quad (1)$$

such that $max_j$ (respectively $min_j$) is the observed *maximum* (respectively *minimum*) value of the output $o_j$. Given a test suite $TS$, suppose we intend to extend $TS$ with another test suite $TS_{cand}$. The output diversity test objective, denoted by $fit_{od}(TS_{cand} \cup TS)$, is defined as follows:

$$fit_{od}(TS_{cand} \cup TS) = Min \left\{ dist(O_{tc_i}, O_{tc_j}) \right\}_{\forall tc_i \in TS_{cand} \wedge \forall tc_j \in TS \cup TS_{cand} \wedge j \neq i}$$

That is, we compute the minimum distance among all the distances between pairs $tc_i \in TS_{cand}$ and $tc_j \in TS_{cand} \cup TS$. The larger the value of $fit_{od}$ the further apart the output signals produced by test cases in $TS_{cand}$ and $TS$.

**Test generation algorithm.** Having defined the fitness functions, we now define our search-based test generation algorithm (i.e., TESTGENERATION in Fig. 3). Two versions of the TESTGENERATION algorithms are shown in Fig. 4 and Fig. 5. The algorithm in Fig. 4 generates test cases based on any of our three coverage-based test objectives, i.e., coverage dissimilarity, coverage density, and number of DBB. The one in Fig. 5 is designed to utilize our fourth test objective, i.e., output diversity. These algorithms adapt a single-state search optimizer [46]. In particular, they build on Hill-Climbing with Random Restarts (HCRR) heuristics [46]. We chose HCRR because in our previous work on testing Simulink models [52], HCRR was able to produce the best-optimized test cases among other single-state optimization algorithms. Computation of all the four fitnesses we described earlier rely on either test execution slices or generated output values. To obtain test execution slices or output values, 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. 4** Test case generation algorithm (Coverage Dissimilarity, Coverage Density, and DBB).

The algorithm in Fig. 4 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 [53]. (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).

The algorithm presented in Fig. 5 differs from the algorithm in Fig. 4 as follows: since with our fourth test objective, i.e., output diversity, computation is based on

**Algorithm.** TESTGENERATION(OUTPUT DIVERSITY)
**Input:** - $OUT_{TS}$: The set of output vectors for current test suite
- $M$ : 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. $OUT_{curr} \leftarrow$ Generate the set of the output vectors of the $k$ test cases in $TS_{curr}$
3. $fit_{curr} \leftarrow$ ComputeFitness $(OUT_{curr}, OUT_{TS}, M)$
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. $OUT_{new} \leftarrow$ Generate the set of the output vectors of the $k$ test cases in $TS_{new}$
9. $fit_{new} \leftarrow$ ComputeFitness $(OUT_{new}, OUT_{TS}, M)$
10. **if** ($fit_{new} > fit_{curr}$ )
11. $fit_{curr} \leftarrow fit_{new}$; $TS_{curr} \leftarrow TS_{new}$
12. **end**
13. **if** ($fit_{curr} \geq fit_{best}$ )
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. 5** Test case generation algorithm (Output Diversity).

the output vectors, we do not collect test execution slices. Instead, we gather output vectors generated for each test case (i.e., $OUT_{curr}$ and $OUT_{new}$).

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 [23]. 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. The computation of $fit_{OD}$ requires output vectors instead of test execution slices. Test oracle information for the $k$ new test cases is only needed after test generation in subroutine STATISTICALDEBUGGING (see Fig. 3) when a new statistical ranking is computed. In the next section, we discuss the STOPTESTGENERATION subroutine (see Fig. 3) 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.

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. $m_3$ = *ComputeRankCorrelation*(*initialList*, *newList*)
10. Build a *prediction model* based on a subset of $\{m_1, m_2, m_3, r\}$ and let *result* be the output of the *prediction model*.
11. **return** *result*

**Fig. 6** The STOPTESTGENERATION subroutine used in our approach (see Fig. 3).

### 3.2 Stopping Test Generation

As noted in the literature [17], 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 Fig. 6. 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 7-11), 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* [14]) based on the following four features: (1) the current test generation round, (2) the *SetDistance* between the latest ranked list and the initial ranked list, (3) the *OrderingDistance* between the latest ranked list and the initial ranked list, and (4) the *RankCorrelation* between the latest ranked list and the initial ranked list. Below, we first introduce Simulink super blocks. We will then introduce *SetDistance*, the *OrderingDistance* and the *RankCorrelation* 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 3.1. The only difference is that dynamic basic blocks are defined with respect to the test execution slices generated by a given test suite, while super blocks are defined with respect to test execution slices that can be generated by any potential test case. Hence, dynamic basic blocks

can be computed *dynamically* based on test execution slices obtained by the current test suite, whereas super blocks are computed by *static analysis* of the structure of Simulink models. In order to compute super blocks, we identify conditional (control) blocks in the given Simulink model. Each conditional block has an incoming control link and a number of incoming data links. Corresponding to each conditional block, we create some branches by matching each incoming data link with the conditional branch link. We then remove the conditional block and replace it with the new branches. This allows us to obtain a behaviorally equivalent Simulink model with no conditional blocks. We further remove parallel branches by replacing them with their equivalent sequential linearizations. We then use the resulting 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 four features used in our predictor models. The first feature is the test generation round. As shown in Fig. 3, 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, third and fourth features (i.e., *SetDistance*, *OrderingDistance*, and *RankCorrelation*) are similarity metrics comparing the latest generated rankings (at the current round) and the initial rankings. These three 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* [22]. We focus on comparing the top $N$ ranked groups because, in practice, the top ranked groups are primarily inspected by engineers. We compute the *SetDistance* based on the average of the overlap between the top-$N$ ranked groups of the two ranked lists. Formally, we define the *SetDistance* between $initialList$ and $newList$ as follows.

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

$$\textit{SetDistance}(initialList, newList) = 1 - IM(initialList, newList)$$

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* [36] 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:

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

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

Our fourth feature is *RankCorrelation*. Similar to *OrderingDistance*, *RankCorrelation* aims to measure differences between the top-N ranked groups of *initialList* and *newList*. However, *OrderingDistance* depends on the number of discordant pairs, while *RankCorrelation* depends on degrees of ranking changes between *initialList* and *newList* for individual elements. In particular, we define *RankCorrelation* based on *Spearman's Rank Correlation Coefficient (Ties-corrected)* [37, 67] that is a well-known metric for comparing ordered lists. In our work, to compute the correlation between two rankings (*initialList* and *newList*), we first create a set *unionSet* based on *initialList* and *newList*: *unionSet* contains all the elements that appear in the top-N rank groups in both *initialList* and *newList*. Then, we compute two new lists *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 *unionSet* are removed. Similarly, *newL* is the same as *newList* except that all the blocks that do not appear in *unionSet* are removed. Due to removal of elements from $newL$ and $initialL$, we adjust the ranks for elements in $initialL$ and $newL$ so that there is no gap between the rank values in these two lists. Then, for each element $e_i$ in *unionSet*, we define the distance $d_i$ to be the difference between the rank of $e_i$ in *initialL* and the rank of $e_i$ in *newL*. Let the size of $unionSet$ be $n$ (i.e., $|unionSet|$ = $n$). We define the *RankCorrelation* metric as follows:

$$RankCorrelation = 1 - \frac{6\left(\sum_{i=1}^{n} d_i^2 + \frac{1}{12}CF\right)}{n(n^2-1)}$$

In the above formula, $CF$ is referred to as a *correction factor*. This factor is proposed to adapt the original computation of *Spearman correlation coefficient* to situations, like our case, where rankings contain rank groups with multiple elements [37]. We denote by $rg_i$ (respectively, $rg'_i$) the $i$th rank group in *initialL* (respectively, *newL*). We further denote by $G$ (respectively, $G'$) the total number of rank groups in *initialL* (respectively, *newL*). Then, according to [37], $CF$ is computed as follows:

$$CF = \sum_{0 \leqslant i \leqslant G} |rg_i| * (|rg_i|^2 - 1) + \sum_{0 \leqslant j \leqslant G'} |rg'_j| * (|rg'_j|^2 - 1)$$

The value range of *RankCorrelation* is $[-1, 1]$. The larger the *RankCorrelation*, the more similar the top-N ranked groups of *initialList* and *newList* are.

**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 different combinations of the four features discussed above (i.e., the current round, *SetDistance*, *OrderingDistance*, *RankCorrelation*). We use supervised learning methods, and in particular, decision trees [14]. 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 (Fig. 3) 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 (Fig. 3), and suppose $initList$ is a ranking obtained initially before generating test cases (Fig. 3). 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 Fig. 3, we obtain a tuple $I$ consisting of a subset of our four features defined above. 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 Fig. 3. We denote the maximum fault localization accuracy improvement by $Max\_ACC_r(rankList)$. We then label the tuple $I$ 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.

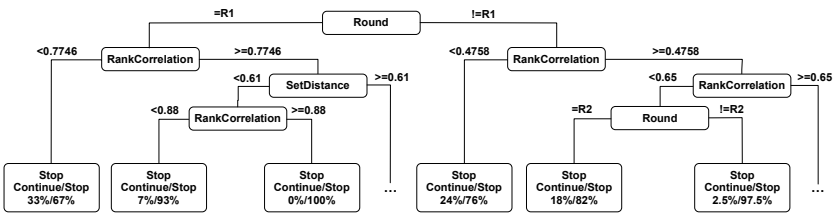


**Fig. 7** A snapshot example of a decision tree built based on the following input features: Round, *SetDistance*, and *RankCorrelation*).

Having obtained tuples $I$ labeled 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 deci-

sion tree model is built by partitioning the set of input tuples in a stepwise manner aiming to create partitions with increasingly more homogeneous labels (i.e., partitions in which the majority of tuples are labeled either by `Stop` or by `Continue`). The larger the difference between the number of tuples with `Stop` and `Continue` in a partition, the more homogeneous that partition is. Decision variables in our decision tree model represent logical conditions on the input features $r$, *SetDistance*, *OrderingDistance*, or *RankCorrelation*. Fig. 7 shows a fragment of our decision tree model built based on the features: Round, *SetDistance* and *RankCorrelation*. For example, this model shows, among the tuples satisfying $r$ = `R1` and *RankCorrelation* $< 0.7746$ conditions, 67% are labeled with `Stop` and 33% are labeled with `Continue`. As we will discuss in Section 4.2, we experiment with seven different combinations of our four features to identify the most effective prediction model. In these combinations, we do not select *RankCorrelation* and *OrderingDistance* together since these two features represent two alternative ways measuring similarities of a pair of ordered lists.

We stop splitting partitions in our decision tree model if the number of quadruples in the partitions is smaller than $\alpha$, or the percentage of the number of quadruples in the partitions with the same label is higher than $\beta$. In this work, we set $\alpha$ to 50 and $\beta$ 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*,*OrderingDistance* and *RankCorrelation* features corresponding to $newList$. We then build a prediction model based on a subset of these three features as well as $r$ (i.e., the round). 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 labeled `Stop`. Otherwise, our prediction model returns *false*, indicating that test generation should be continued. For example, assuming the decision tree in Fig. 7 is our prediction model, we stop test generation only if we are not in round one, *RankCorrelation* is greater than or equal to $0.88$, and *SetDistance* is less than $0.61$. This is because, in Fig. 7, these conditions lead to the leaf partition with 100% stop-labeled elements.

## 4 Empirical Evaluation

In this section, we empirically evaluate our approach using experiments applied to real-world Simulink models from the automotive domain.

### 4.1 Research Questions

**RQ1. [Evaluating and comparing different test generation fitness heuristics]** *How is the fault localization accuracy impacted when we apply our search-based test generation algorithm in Fig. 4 with our four selected fitness functions (i.e., coverage dissimilarity ($f_{Dis}$), coverage density ($f_{Dens}$), number of dynamic basic blocks ($f_{dbb}$)), and output diversity ($f_{od}$)?* 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 Fig. 4 with a small number of test cases. We further compare the fault localization accuracy improvement when we use our four alternative fitness functions, and when we use a random test generation strategy not guided by any of these fitness functions.

**RQ2. [Evaluating and comparing different generation methods for the initial test suite]** *What is the impact of the technique used for generation of the initial test suite on fault localization results?* To answer this question, we compare three different test generation techniques, namely *Adaptive Random Testing*, *Coverage-based*, and *Output Diversity*. *Adaptive Random Testing* technique is a typical baseline test generation technique that aims to increase the diversity of test cases by maximizing distances between the test inputs. *Coverage-based* techniques attempt to generate test suites that are able to achieve a high degree of structural coverage over Simulink models under tests [53]. Recall that *Output Diversity* is described in Section 3.1. Our goal is to see if the generation methods employed to create initial test suites for debugging our Simulink models have an impact on the effectiveness of our fault localization results.

**RQ3. [Evaluating impact of adding test cases]** *How does the fault localization accuracy change when we apply our search-based test generation algorithm in Fig. 4?* We note that adding test cases does not always improve the fault localization accuracy [17]. With this question, we investigate how often fault localization accuracy improves after adding test cases. In particular, we apply our approach in Fig. 3 without calling the STOPTESTGENERATION subroutine, and identify how often subsequent rounds of test generation do not lead to fault localization accuracy improvement.

**RQ4. [Effectiveness of our STOPTESTGENERATION subroutine]** *Among the features introduced in Section 3.2, which feature combination yields the best results for our* STOPTESTGENERATION *subroutine? Does our* STOPTESTGENERATION *subroutine, when used with the best performing feature combination, help stop generating additional test cases that do not improve the fault localization accuracy?* We first compare the performance of alternative prediction models built based on different feature combinations discussed in Section 3.2. Our goal is to identify the feature combination that yields the best trade-off between fault localization accuracy and the number of newly generated test cases. We then investigate whether the predictor models built based on the best feature combination 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.

**RQ5. [Impact of the threshold parameter on predictor models]** *Does the effectiveness of the* STOPTESTGENERATION *subroutine depend on the threshold parameter used to label the training data for the prediction models? How is the effectiveness of our* STOPTESTGENERATION *subroutine affected when we vary the threshold values for labeling the training data?* As discussed in Section 3.2, the $THR$ value is a key parameter for our prediction models and is used to label the training data. In this question, we identify, for a fixed feature combination, the optimal values for $THR$ that yield the best trade-off between fault localization accuracy and the number of newly generated test cases.

## 4.2 Experiment Settings

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

**Industrial Subjects.** In our experiment, we use three Simulink models referred to as *MA*, *MZ* and *MGL*, and developed by Delphi Automotive Systems [21], our industrial partner. Table 2 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 [51].

**Table 2** 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 senior Delphi test engineer to seed 20 realistic and typical faults into each model. The seeded faults belonged to the following fault patterns:

- Wrong arithmetic operators, e.g., replacing a + operator with - or $\times$.
- Wrong relational operators, e.g., replacing a $\leq$ with $\geq$, or $= \neq$.
- Wrong constant values, e.g., replacing constant $c$ with $c - 1$ or $c + 1$.
- Variable Replacement, e.g., replacing a "double" variable with a "single" variable.
- Incorrect connections, e.g., switching the input lines of the "Switch" block.

The above fault patterns represent the most common faults we observed in practice, and are also used in existing literature on mutation operators for Simulink models [16, 24, 28, 78, 81]. In total, we generated 60 faulty versions (one fault per each faulty version). The engineer seeded faults based on his past experience in Simulink development and, to achieve diversity in terms of the location and types of faults, we required faults of different types to be seeded in different parts of the models.

Table 3 shows the number of faulty versions related to each fault pattern in our experiments. Finally, we have also provided detailed descriptions of the seeded faults and all experiment data and scripts at [40].

**Table 3** Number of Fault Patterns Applied to Each Industrial Subject

| | | MA | MZ | MGL |
|---|---|---|---|---|
| | # of Faulty Version | 20 | 20 | 20 |
| Wrong arithmetic operators | # of Faulty Version | 7 | 9 | 5 |
| Wrong relational operators | # of Faulty Version | 1 | 1 | 0 |
| Wrong constant values | # of Faulty Version | 6 | 5 | 8 |
| Variable Replacement | # of Faulty Version | 5 | 4 | 5 |
| Incorrect connections | # of Faulty Version | 1 | 1 | 2 |

Finally, for each faulty model, we randomly generated a large number of input signals, compared the outputs of the faulty model with those of the non-faulty model to ensure that each faulty model exhibits some visible failure in some model output. We further manually inspected each faulty model whose outputs deviated from some outputs of the non-faulty model to convince ourselves that the two models are not behaviorally equivalent, and that the seeded fault led to a visible failure in some model output.

**Initial Test Suites and Test Oracles.** In our experiments, we use three different test generation techniques, namely *Adaptive Random Testing*, *Coverage-based*, and *Output Diversity*, to generate the initial test suites.

1. **Adaptive Random Testing**: Adaptive random testing [18] 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 inputs.

**Algorithm.** ADAPTIVE RANDOM TESTING
**Input:** - $RG$ : valid value range of each input signal
- $M$ : The Simulink model
**Output:** $TS$: A set of test cases

1. $TS$=$\{I\}$, where $I$ is a randomly-generated test cases of $M$
2. **for** ($q - 1$ times ) **do:**
3. $MaxDist$ = 0
4. Let $C = \{I_1, ..., I_c\}$ be a candidate set of random test cases of $M$
5. **for each** $I_i \in C$ **do:**
6. $Dist$=$MIN_{\forall I' \in TS} dist(I_i, I')$
7. **if** ($Dist > MaxDist$)
8. $MaxDist = Dist$, $J = I_i$,
9. $TS = TS \cup J$
10. **return** $TS$

**Fig. 8** Initial test suite generation (Adaptive Random Testing).

Fig. 8 shows the Adaptive Random Testing algorithm that, given a Simulink model $M$ and the valid signal range of each input, generates an initial test suite

$TS$ with size $q$. The algorithm first randomly generates a single test case and stores it in $TS$ (line 1). Then at each iteration, it randomly generates $c$ candidates test cases $I_1$, ..., $I_c$. It computes the distance of each test case $I_i$ from the existing test suite $TS$, and select the test case with the minimum distance between $I_i$ and the test cases in $TS$ (line 6). Finally, the algorithm identifies and adds the candidate test case with the maximum distance from $C$ into $TS$ (line 7-9).

2. **Coverage-based**: We consider a block coverage criterion for Simulink models that is similar to *statement* coverage for programs [49] and *state* coverage for Stateflow models [13]. Our coverage-based test generation algorithm is shown in Fig. 9. In line 1, the algorithm selects a random test input $I$ and adds the corresponding model coverage to a set $TSCov$. At each iteration, the algorithm generates $c$ candidate test cases and computes their corresponding model coverage information in a set $CCov$. It then computes the additional coverage that each one of the test cases in $C$ brings about compared to the coverage obtained by the existing test suite $TS$ (line 8). At the end of each iteration, the test case that leads to the maximum additional coverage is selected and added into $TS$ (line 13). Note that if none of the $c$ candidates in $C$ yields an additional coverage, i.e., $MaxAddCov$ is 0 at line 11, we pick a test case with the maximum coverage in $C$ (line 12).

**Algorithm.** COVERAGE-BASED
**Input:** - $RG$ : valid value range of each input signal
- $M$ : The Simulink model
**Output:** $TS$: A set of test cases

1. $TS$=$\{I\}$, where $I$ is a randomly-generated test cases of $M$
2. $TSCov$=$\{Cov\}$, where $Cov$ is the model coverage information of exeucting $M$ with $I$
3. **for** ($q - 1$ times ) **do:**
4. $MaxAddCov$ = 0
5. Let $C = \{I_1, ... , I_c\}$ be a candidate set of random test cases of $M$
6. Let $CCov = \{Cov_1, ... , Cov_c\}$ be the coverage information of executing $M$ with $C$
7. **for each** $Cov_i \in CCov$ **do:**
8. $AddCov$=$|Cov_i - \cup_{S' \in TSCov} S'|$
9. **if** ($AddCov > MaxAddCov$)
10. $MaxAddCov$ = $AddCov$, $J$ = $I_i$, $P$ = $Cov_i$
11. **if** ($MaxAddCov$ = 0)
12. $J = I_j$, $P = Cov_j$ where $Cov_j \in CCov$ and $|Cov_j| = MAX_{Cov' \in CCov}|Cov'|$
13. $TS = TS \cup J$, $TSCov = TSCov \cup P$
14. **return** $TS$

**Fig. 9** Initial test suite generation (Coverage-based).

3. **Output Diversity**: To generate an initial test suite based on the notion of output diversity, we use the algorithm shown in Fig. 5 by setting $k$ to 1.

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. Specifically,

we generate an initial test suite of size 10 for each of the test generation methods discussed above. To enable the full automation of our experiments, we used the fault-free versions of our industrial subjects as test oracles.

**Experiment Design.**

To answer **RQ1** to **RQ5**, we performed four experiments **EXP-I** to **EXP-IV** described below:

**EXP-I** focuses on answering **RQ1** and **RQ3**. Fig. 10(a) shows the overall structure of **EXP-I**. We refer to the test generation algorithm in Figs. 4 and 5 as HCRR since they build on the HCRR search algorithm. We refer to HCRR when it is used with test objectives $f_{Dis}$, $f_{Dens}$, $f_{dbb}$, and $f_{od}$ as HCRR-Dissimilarity, HCRR-Density, HCRR-DBB and HCRR-OD, respectively. We set both the number of new test cases per round (i.e., $k$ in Fig. 3), and the number of rounds (i.e., $round$ in Fig. 3) to five. That is, in total, we generate $25$ new test cases by applying our approach. We apply our four alternative HCRR algorithms, as well as the Random test generation algorithm, which is used as a baseline for our comparison, to our $60$ faulty versions. We ran each HCRR algorithm for 45 minutes with two restarts. To account for the randomness of the search algorithms, we repeat our experiments for ten times (i.e., ten trials). Note that the initial test suite (i.e., $TS$ in Fig. 3) contains ten test cases generated by using *Adaptive Random Testing* technique.

**EXP-II** answers **RQ2** and evaluates the impact of different initial test suites on the fault localization accuracies. We use the best test objective (i.e., *HCRR-DBB*) based on the comparison result of **RQ1**. Fig. 10(b) shows the overall structure of **EXP-II**. To answer **RQ2**, we repeat the experiment **EXP-I** while we only use one test objective (i.e., *HCRR-DBB*) with three different initial test suites generated by the three different test generation techniques, i.e., *Adaptive Random Testing*, *Coverage-based*, and *Output Diversity*, described earlier in this section.

**EXP-III** answers the research question **RQ4** and evaluates the effectiveness of our STOPTESTGENERATION subroutine. Fig. 10(c) shows the overall structure of **EXP-III**. Based on the features we defined in Section 3.2, we built different combinations of features for our prediction models. We identified, in total, the following seven input feature sets:

$FC_1 = \{r, SetDistance, OrderingDistance\}$
$FC_2 = \{r, SetDistance, RankCorrelation\}$
$FC_3 = \{r, SetDistance\}$
$FC_4 = \{r, OrderingDistance\}$
$FC_5 = \{r, RankCorrelation\}$
$FC_6 = \{SetDistance, OrderingDistance\}$
$FC_7 = \{SetDistance, RankCorrelation\}$

We evaluate and compare different prediction models built based on these seven input feature combinations. We set the $THR$ parameter to 15. Recall from Section 3.2 that $THR$ is the threshold parameter used to label the training data for creating prediction models.

**EXP-IV** answers the research question **RQ5**. Fig. 10(d) shows the overall structure of **EXP-IV**. In the **EXP-IV**, we repeat the experiment **EXP-III** by using the best input feature set from **RQ4** (i.e., $FC_1 = \{r, SetDistance, OrderingDistance\}$), but we vary $THR$ values by setting it to 5, 10, 15, 20, 25, 30, and 35, respectively.

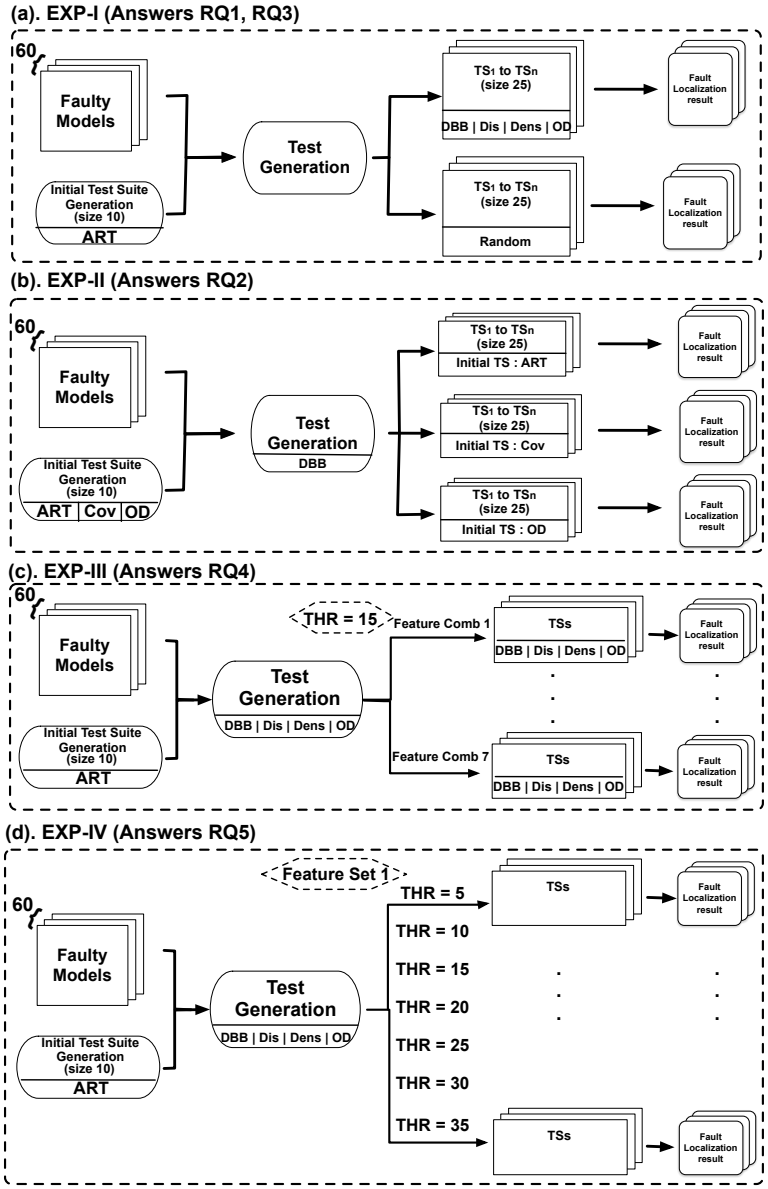


**Fig. 10** Our experiment design: (a) **EXP-I** to answer **RQ1**. (b) **EXP-II** to answer **RQ2**. Test generation algorithms are repeated for 10 times to account for their randomness for **EXP-I** and **EXP-II**. (c) **EXP-III** to answer **RQ4**. (d) **EXP-IV** to answer **RQ5**.

We ran our experiments on a high performance computing platform [71] with 2 clusters, 280 nodes, and 3904 cores. Our experiments 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 13500 hours. Most of the experiment time was used to execute the generated test cases in Simulink. In total, we generated and executed 299000, 290000, and 323000 test cases for *MA MZ*, and *MGL*, respectively.

**Table 4** The wilcoxon test results ($p$-values) and the $A_{12}$ effect size values comparing distributions in Fig. 11(a).

| Pair<br>A vs. B | p-value | $A_{12}$ |
|---|---|---|
| HCRR-DBB vs. Initial | 0.00 | 0.73 |
| HCRR-DBB vs. Random | 0.00 | 0.65 |
| HCRR-Density vs. Initial | 0.00 | 0.72 |
| HCRR-Density vs. Random | 0.00 | 0.65 |
| HCRR-Dissimilarity vs. Initial | 0.00 | 0.73 |
| HCRR-Dissimilarity vs. Random | 0.00 | 0.66 |
| HCRR-OD vs. Initial | 0.00 | 0.7 |
| HCRR-OD vs. Random | 0.00 | 0.62 |

### 4.3 Evaluation Metrics

We evaluate the accuracy of the rankings generated at different rounds of our approach using the following metrics [20, 33, 44, 45, 57, 62]: 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 3.2. The proportion of faults localized is the proportion of localized faults over the total number of faults when engineers inspect a fixed number of the top most suspicious blocks from a ranking.

### 4.4 Experiment Results

**RQ1. [Evaluating and comparing different test generation fitness heuristics]** To answer this question, we performed **EXP-I**. Fig. 11 compares the fault localization results after applying HCRR-DBB, HCRR-Density, HCRR-Dissimilarity and HCRR-OD 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 Fig. 11(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, HCRR-OD and Random test generation to add 25 test cases to the initial test suite. Each point in Fig. 11(a) represents fault localization accuracy for one run of one faulty version. According to Fig. 11(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, 42.8 and 46.4 blocks if they use the rankings generated by HCRR-DBB, HCRR-Density, HCRR-Dissimilarity and HCRR-OD, respectively. We performed non-parametric pairwise Wilcoxon signed-rank test to check whether the improvement on the number of blocks inspected is statistically significant. We also computed Vargha and Delaney's $\hat{A}_{12}$ [70] to compare fault localization results

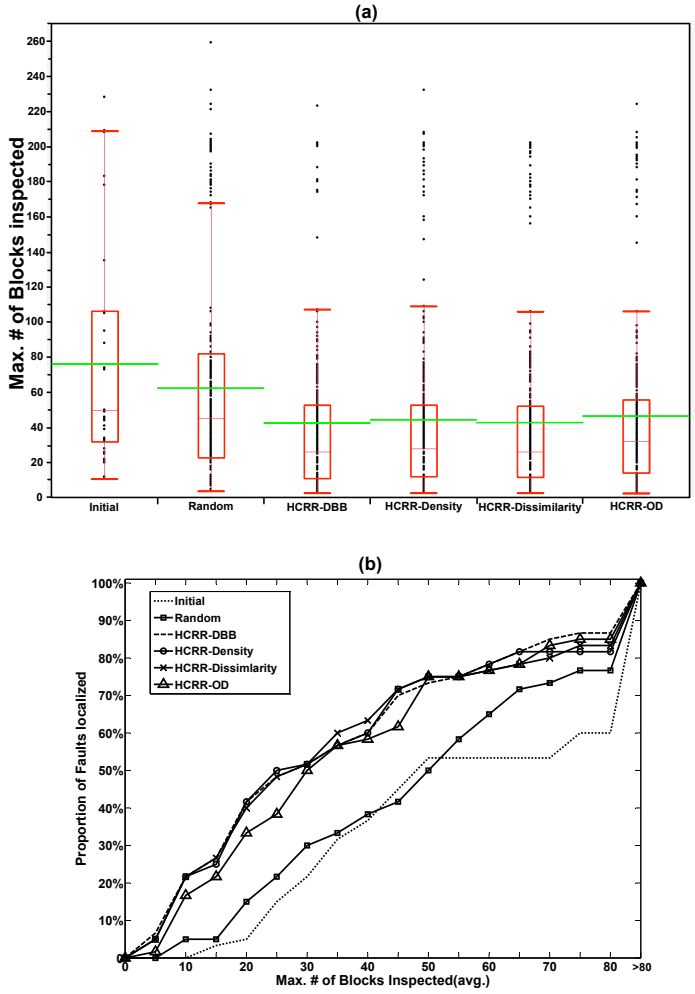


**Fig. 11** Comparing the number of blocks inspected (a) and the proportion of faults localized (b) before and after applying HCRR-DBB, HCRR-Dissimilarity, HCRR-Density and HCRR-OD, and with Random test generation (i.e., Random).

reported in Figure 11(a). The statistical test $p$-values and the effect size values comparing the distributions in Figure 11(a) are reported in Table 4. We note that each distribution in Figure 11(a) consists of 600 points (i.e, 60 faulty versions $\times$ 10 runs). The results in Table 4 show that the fault localization accuracy distributions obtained by HCRR-DBB, HCRR-Density, HCRR-Dissimilarity and HCRR-OD are significantly lower (better) than those obtained by Random and Initial (with $p$-value$<$0.0001). As for the effect size results, two algorithms are considered to be equivalent when the value of $\hat{A}_{12}$ is 0.5. The closer the effect size values to 1 for each comparison (Algo A vs. Algo B) the better Algo A compared to Algo B.

Similarly, Fig. 11(b) shows the proportion of faults localized when engineers inspect a fixed number of blocks in the rankings generated by Initial, and after generating 25 test cases with HCRR-DBB, HCRR-Density, HCRR-Dissimilarity, HCRR-OD, 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 Fig. 11(b), the maximum number of blocks inspected (X-axis) is computed as an average over ten trials for each faulty version. According to Fig. 11(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 three of our techniques i.e., HCRR-DBB, HCRR-Density and HCRR-Dissimilarity, and 10 out of 60 (16.67%) faulty versions when they inspect at most 10 blocks in the rankings generated by the fourth technique, i.e., HCRR-OD. 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, HCRR-Dissimilarity and HCRR-OD, 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 44 blocks (i.e., 42.1% reduction). Further, we have shown that the fault localization accuracy results obtained based on HCRR-DBB, HCRR-Density, HCRR-Dissimilarity and HCRR-OD 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 44 blocks when HCRR-DBB, HCRR-Density, HCRR-Dissimilarity and HCRR-OD are used.

**RQ2. [Evaluating and comparing different test generation methods for the intitial test suite]** To answer **RQ2**, we applied **EXP-II** and obtained the fault localization results based on the initial test suites generated by Adaptive Random Testing (ART), Coverage-based (Cov) and Output Diversity (OD) as well as the fault localization results after extending the initial test suites with new test cases using our search-based test generation algorithm.

Figs. 12 (a) - (d) show the results of the **EXP-II** experiment. Figs. 12(a) and (c) show the fault localization results obtained based on different initial test suites *before* applying our search-based test generation technique, and Figs. 12(b) and (d) show the fault localization results obtained based on different initial test suites *after* applying our search-based test generation technique. Specifically, Figs. 12(a) and (b) show the number of blocks needed to be inspected to identify the faulty blocks, and Figs. 12(c) and (d) report the proportion of faults that can be localized when inspecting a fixed number of blocks in rankings. Note that, in this experiment, we use HCRR-DBB to generate additional test cases (i.e., the best test objective according to **RQ1**).

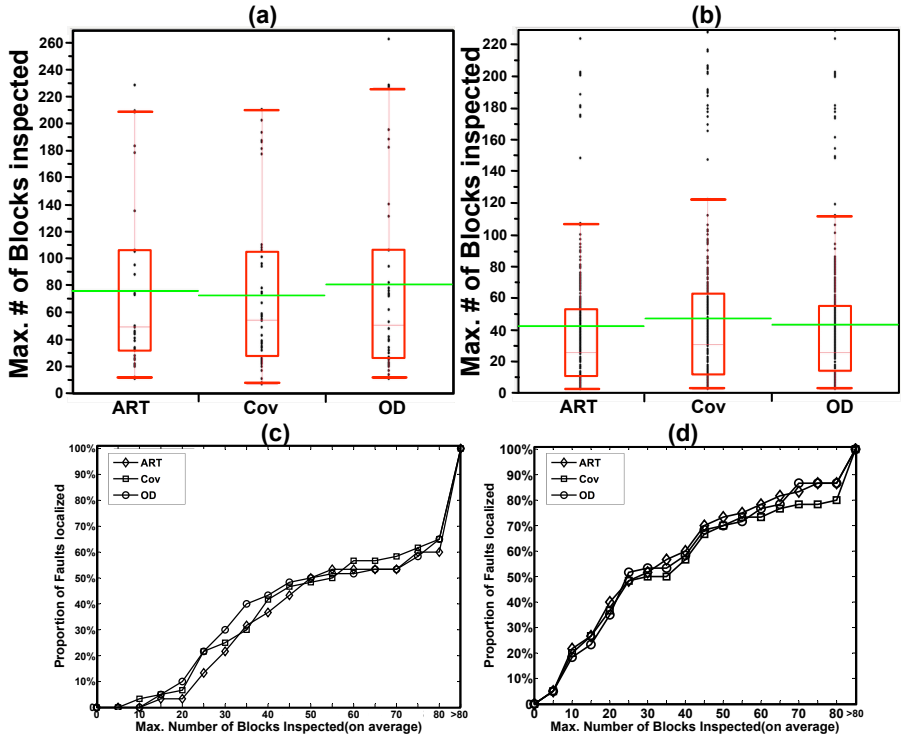


**Fig. 12** Comparing the fault localization accuracy results obtained based on different initial test suites generated by Adaptive Random Testing (ART), Coverage-based (Cov), and Output Diversity (OD): (a) the number of blocks inspected before adding new test cases; (b) the number of blocks inspected after adding new test cases; (c) the proportion of faults localized before adding new test cases; and (d) the proportion of faults localized after adding new test cases.

According to Fig. 12(a), engineers on average need to inspect at most 76, 72, and 80 blocks to locate faults when they generate rankings using the initial test suites generated by ART, Cov, and OD, respectively. As shown in Fig. 12(b), after adding new test cases, on average, the numbers of blocks engineers need to inspect reduce to 42.4, 47.2, and 43.2 for the initial test suites generated by ART, Cov, and OD, respectively. As these diagrams show, the differences between the fault localization results computed based on different initial test suites before and after applying our search-based test generation algorithm are small. Similarly, as Figs. 12 (c) and (d) show, the differences between the proportions of faults localized when a fixed number of blocks are inspected are very close for different initial test suites and considering results before and after applying test suite expansion.

We further applied the Wilcoxon test and computed the $A_{12}$ effect size values to statistically compare the effectiveness of our approach for different initial test suites. The distributions that we statistically compare each have 600 points (i.e., each distribution in Figs. 12(a) and (b) has 600 points). The $p$-values and the effect size values are shown in Table 5. Specifically, the table shows the $p$-values (the $A_{12}$ effect size values resp.) obtained by comparing the pairs of distributions in Fig. 12 (a) together (i.e., those obtained before test generation), and comparing the pairs of distributions

in Fig. 12 (b) together (i.e., those obtained after test generation). As shown in Table 5, the differences between fault localization results obtained by different initial test suites are not statistically significant, i.e., all $p$-values are higher than 1%, our chosen level of significance ($\alpha$). Further, the effect size values are close to 0.5.

**Table 5** The wilcoxon test results (p-values) and the $A_{12}$ effect size values comparing pairs of distributions in Fig. 12. The cells show: (the p-value for before test generation results) / (the p-value for after test generation results) or (the $A_{12}$ effect size for before test generation results) / (the $A_{12}$ effect size for after test generation results).

| Pair<br>A vs. B | $p$-value | $A_{12}$ |
|---|---|---|
| ART vs. OD | 0.58 / 0.06 | 0.53/0.49 |
| ART vs. Cov | 0.14 / 0.011 | 0.52/0.48 |
| Cov vs. OD | 0.015 / 0.07 | 0.5/0.51 |

*In summary*, among the three test generation techniques, Cov leads to the best fault localization accuracy average before expanding initial test suites, and ART yields the best fault localization accuracy average after adding new test cases. However, in general, the differences among fault localization accuracies obtained based on different initial test suites are not statistically significant.

**RQ3. [Evaluating impact of adding test cases]** To answer this question, we evaluate the fault localization accuracy of the ranking results obtained at each test generation round. These results produced by performing **EXP-I**. In particular, we computed the fault localization accuracy of rankings obtained by applying HCRR-DBB, HCRR-Density, HCRR-Dissimilarity and HCRR-OD 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 2400 trials (60 faulty versions $\times$ 4 algorithms $\times$ 10 runs). Among these 2400 trials, we observed that, as we go from round one to round five, in 1376 cases (i.e., 57.3%), the fault localization accuracy improves at every round; in 952 cases (i.e., 39.7%), the accuracy improves at some (but not all) rounds; and in 72 cases (i.e., 3%), 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 [42]. 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.

**RQ4. [Effectiveness of our STOPTESTGENERATION subroutine]** To answer this question, we performed **EXP-III**. In order to generate alternative prediction models for the STOPTESTGENERATION subroutine, we proposed four features and cre-

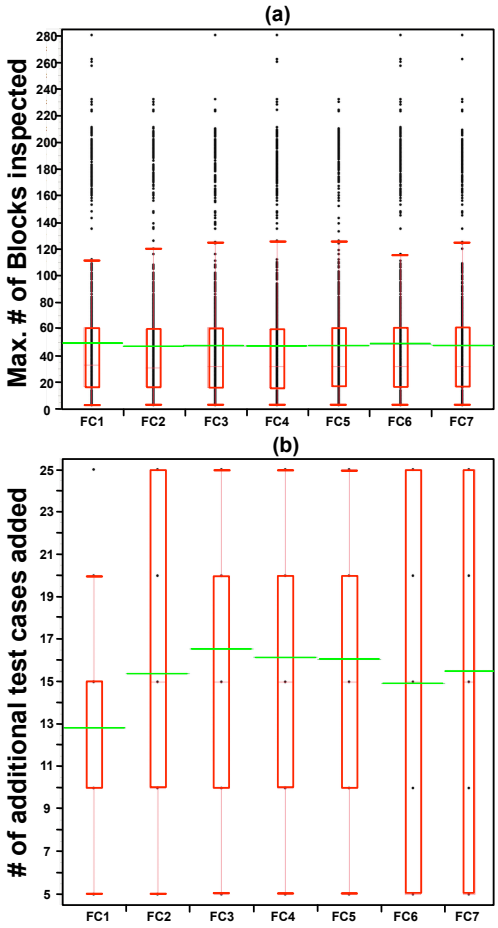


**Fig. 13** Comparing the performance of STOPTESTGENERATION with predictor models built based on different feature combinations $FC_1$ to $FC_7$: (a) The maximum number of blocks inspected, and (b) the number of new test cases added.

ated seven different input feature combinations (see Section 4.2 and Fig. 10 (c)) for our prediction models. In this question, 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. For all of these rankings, we randomly divide them 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 3.2). That is, engineers are willing to undergo the overhead of adding new test cases if the fault localization accuracy is likely to improve by at least 15 blocks.

To answer which feature combination among the seven alternative combinations performs the best, we build alternative prediction models based on seven different

feature combinations. Fig. 13 (a) shows *the fault localization accuracy results* (i.e., the maximum number of blocks inspected) obtained by our test generation algorithms when the prediction model used in the STOPTESTGENERATION subroutine is built based on the seven alternative feature combinations. Fig. 13 (b) shows *the number of new test cases* generated by our search-based algorithm when we applied the STOPTESTGENERATION subroutine using prediction models built based on the seven feature combinations. Note that we generate 25 test cases if we do not use the STOPTESTGENERATION subroutine at all.

Our goal is to identify the feature combination that yields the best fault localization accuracy with the fewest number of newly generated test cases. Considering that developing test oracles is expensive, and given that the fault localization accuracies for the seven feature combinations are very close (see Fig. 13 (a)), the feature combination $FC_1 = \{r, SetDistance, OrderingDistance\}$ is the best performing one. Specifically, $FC_1$ achieves almost the same fault localization accuracy as the other combinations, but leads to significantly fewer newly generated test cases (i.e., an average of 12.8 test cases).

We further computed the statistical test results and the $A_{12}$ effect size values to compare different feature combinations with respect to the number of additional test cases generated (i.e., comparing distributions in Fig. 13(b)). We note that each distribution in Fig. 13(b) consists of 4800 points. The resulting $p$-values and effect size values are reported in Table 6. Specifically, the table shows that the number of new test cases generated by $FC_1$ is significantly less than those created by other feature combinations with medium effect size values.

**Table 6** The $p$-values and effect size values comparing the FC1 feature combination with other feature combinations based on the distributions reported in Fig. 13 (b).

| Pair A vs. B | $p$-value | $A_{12}$ |
|---|---|---|
| FC1 vs. FC2 | 0.00 | 0.41 |
| FC1 vs. FC3 | 0.00 | 0.34 |
| FC1 vs. FC4 | 0.00 | 0.35 |
| FC1 vs. FC5 | 0.00 | 0.36 |
| FC1 vs. FC6 | 0.00 | 0.46 |
| FC1 vs. FC7 | 0.00 | 0.35 |

We now show the degree of improvement that STOPTESTGENERATION subroutine brings about when it uses a prediction model built based on feature combination $\{r, SetDistance, OrderingDistance\}$.

Fig. 14(a) shows the fault localization accuracy results (i.e., the maximum number of blocks inspected) obtained by our four test generation algorithms (HCRR-DBB, HCRR-Density, HCRR-Dissimilarity and HCRR-OD) 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. Fig. 14(a), further, shows the accuracy results obtained by applying the five rounds *without* using STOPTESTGENERATION in columns labeled `without stop`. In addition, Fig. 14(b) shows the number of new test cases generated by HCRR-

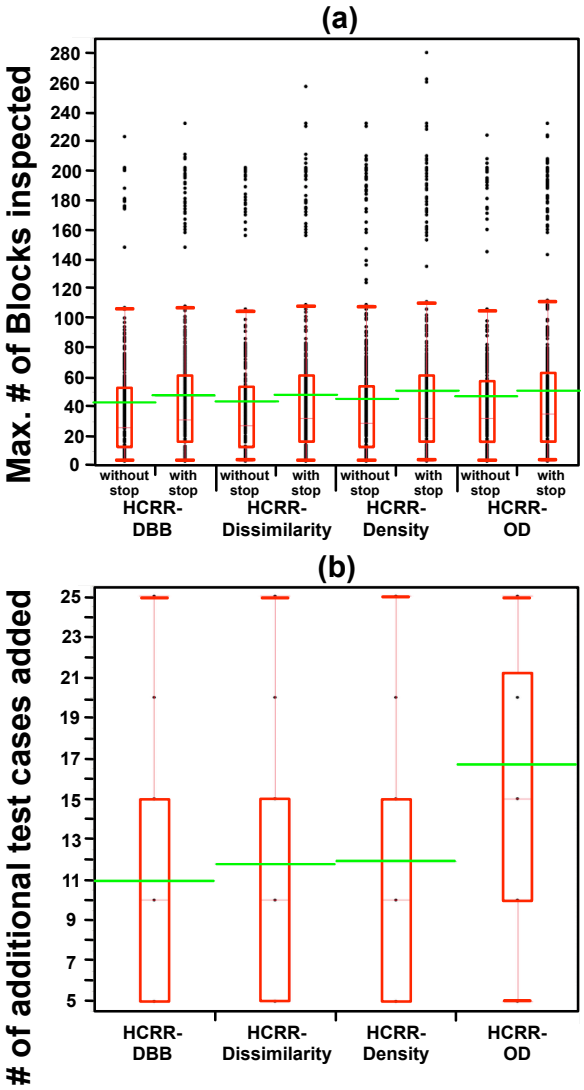


**Fig. 14** Comparing the performance of STOPTESTGENERATION applied with different test objectives HCRR-DBB, HCRR-Density, HCRR-Dissimilarity and HCRR-OD, and with a predictor model built based on the feature combination $FC_1$: (a) The maximum number of blocks inspected, and (b) the number of new test cases added.

DBB, HCRR-Density, HCRR-Dissimilarity and HCRR-OD when we applied the STOPTESTGENERATION subroutine. Note that the results related to $FC_1$ in Fig. 13(b) is the combined results obtained for all the four test generation methods, while in Fig. 14(b), the same results are shown per each test generation method separately.

According to Fig. 14, we are able to obtain almost the same fault localization accuracy with considerably fewer new test cases when we use STOPTESTGENERATION subroutine compared to when we do not use it. In particular, on average, when we use the STOPTESTGENERATION subroutine, the fault localization accu-

racies obtained for HCRR-DBB, HCRR-Dissimilarity, HCRR-Density and HCRR-OD are $47.3$, $47.9$, $50.4$ and $50.8$, respectively. In contrast, without the STOPTESTGENERATION subroutine, the fault localization accuracies obtained for HCRR-DBB, HCRR-Dissimilarity, HCRR-Density and HCRR-OD are $43$, $43.4$, $45.1$ and $47$, respectively. We note that these accuracies are obtained by only generating, on average, 11 test cases for HCRR-DBB, 12 test cases for both HCRR-Density and HCRR-Dissimilarity, and 16.7 test cases for HCRR-OD.

*In summary,* our approach identifies situations where adding new test cases does not improve fault localization results, and helps stop generating additional test cases in such situations. Our experiments show that, among the seven feature combinations described in Section 4.2, a prediction model built based on the feature combination $\{r, SetDistance, OrderingDistance\}$ yields the best trade-off between fault localization accuracy and the number of newly generated test cases. Overall, when engineers use the STOPTESTGENERATION subroutine with the feature combination $\{r, SetDistance, OrderingDistance\}$, they need to inspect a few more blocks (i.e., around five blocks) on average, compared to not using the STOPTESTGENERATION subroutine at all. But the number of test cases, and hence the test oracle cost, reduces significantly (i.e., 33.6% to 56% fewer test cases).

**RQ5. [Impact of predictor model's settings]** To answer this question, we performed *EXP4* as described in Section 4.2 and Fig. 10(d). We evaluate the impact of changing the STOP/CONTINUE threshold, i.e., $THR$, used in STOPTESTGENERATION subroutine.

Fig. 15 shows the results of *EXP4*: In Fig. 15(a), we show the fault localization accuracies (i.e., the maximum number of blocks inspected) obtained by our test generation algorithm when different $THR$ values (i.e., $THR$ = 5, 10, 15, 20, 25, 30, 35) are used to train the prediction models in the STOPTESTGENERATION subroutine. Fig. 15(b) shows the number of newly generated test cases corresponding to different $THR$ values.

As shown in Fig. 15(a), the differences between the fault localization accuracy averages for different $THR$ values are not very large (i.e., less than 11 blocks). In general, by increasing $THR$ values, the number of blocks that needs to be inspected by engineers increases, even though the increase is not very large. On the other hand (as shown in Fig. 15(b)), as we increase $THR$ values from 5 to 35, the average number of newly generated test cases decreases from 19 to 10.

The trends shown in Figs. 15(a) and (b) match our intuition. Specifically, for higher $THR$ values, our STOPTESTGENERATION subroutine stops generating new test cases more quickly. This is because for higher $THR$, this routine requires higher predicted improvement in fault localization accuracy to permit test generation, and hence, in general, it leads to the generation of fewer new test cases. Our results, in addition, show that by increasing $THR$ from 5 to 35, while the number of the newly generated test cases decreases significantly (i.e., around nine fewer test cases on average), the degradation in fault localization accuracy is relatively small (around eleven more blocks to inspect on average).

We also computed $p$-values and effect size values to statistically compare the results obtained by varying the threshold ($THR$) values. Tables 7 and 8 show the

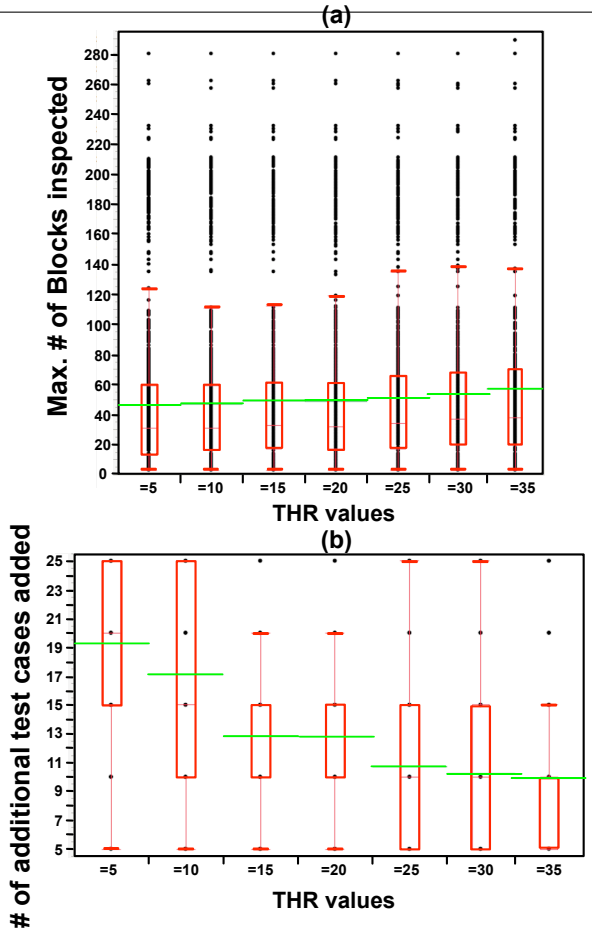


**Fig. 15** Comparing the performance of STOPTESTGENERATION when we vary the $THR$ value from 5 to 35: (a) the maximum number of blocks inspected, and (b) the number of new test cases added.

results comparing the distributions in Figs. 15(a) and (b), respectively. We note that distributions in Figs. 15(a) and (b) each contain 4800 points.

*In summary,* increasing the parameter $THR$ from 5 to 35 reduces both the fault localization accuracy and the number of newly generated test cases significantly. Based on our experiment results, we conclude that an optimal value for $THR$ is between 25 to 30. Compared to $THR$ = 5, with a $THR$ value between 25 to 30, engineers need to inspect four to five more blocks on average, while, on average, are required to develop test oracles for eight to nine fewer test cases.

## 5 Tool Support

We have implemented our fault localization approach in a tool called Simulink Fault Localization tool (*SimFL*). Below, we discuss the working of the tool as well as some information about the implementation of the tool.

Fig. 16 shows an overview of *SimFL*. *SimFL* essentially implements the algorithm of Fig. 3. Specifically, *SimFL* has three main steps: *Statistical Debugging*,

**Table 7** The wilcoxon test results ($p$-values) and the $A_{12}$ effect size values comparing distributions in Fig. 15 (a) (i.e., comparing the impact of threshold values on the number of blocks inspected).

| Pair A vs. B | $p$-value | $A_{12}$ |
|---|---|---|
| THR5 vs. THR10 | 0.02 | 0.50 |
| THR5 vs. THR15 | 0.00 | 0.48 |
| THR5 vs. THR20 | 0.00 | 0.49 |
| THR5 vs. THR25 | 0.00 | 0.47 |
| THR5 vs. THR30 | 0.00 | 0.42 |
| THR5 vs. THR35 | 0.00 | 0.4 |
| THR10 vs. THR15 | 0.00 | 0.49 |
| THR10 vs. THR20 | 0.00 | 0.49 |
| THR10 vs. THR25 | 0.00 | 0.47 |
| THR10 vs. THR30 | 0.00 | 0.42 |
| THR10 vs. THR35 | 0.00 | 0.41 |
| THR15 vs. THR20 | 0.32 | 0.50 |
| THR15 vs. THR25 | 0.01 | 0.49 |
| THR15 vs. THR30 | 0.00 | 0.43 |
| THR15 vs. THR35 | 0.00 | 0.42 |
| THR20 vs. THR25 | 0.00 | 0.48 |
| THR20 vs. THR30 | 0.00 | 0.43 |
| THR20 vs. THR35 | 0.00 | 0.42 |
| THR25 vs. THR30 | 0.00 | 0.44 |
| THR25 vs. THR35 | 0.00 | 0.43 |
| THR30 vs. THR35 | 0.00 | 0.48 |

*Stop Test Generation* and *Test Generation*. These three steps, respectively, match the three subroutines shown in the algorithm of Fig. 3 (i.e., STATISTICALDEBUGGING, STOPTESTGENERATION and TESTGENERATION). We describe the implementation of each of these steps in the *SimFL* tool below.

The *Statistical Debugging* step consists of the following three modules: test case execution, fault localization and super block computation. The test case execution module always precedes the fault localization module. The super block computation module is optional (as indicated by dashed lines in Figure 16). But if this module is activated, it executes before the fault localization module. In the test case execution module, *SimFL* directly invokes Matlab/Simulink to execute the input Simulink model based on the input test suite. The tool then collects test coverage information to compute test execution slices for each individual model output and each test case (i.e., the $TES_{TS}$ set in Fig. 3). The fault localization module receives as input the $TES_{TS}$ set as well as the pass/fail information (oracle) corresponding to each output and each test case. *SimFL* then generates a ranked list using a desired statistical debugging formula specified by the user. Currently, *SimFL* supports two statistical ranking formulas: Tarantula [34] and Ochiai [1]. When the super block computation module is not activated by the user, the ranked list generated by *SimFL* consists of Simulink atomic blocks. Otherwise, if the super block computation module is activated, Simulink super blocks are computed (as discussed in Section 3.2) and are provided as an additional input to the fault localization module. In this case, the ranked list generated by *SimFL* consists of Simulink super blocks.

*SimFL* uses the user interface shown in Fig. 17 to visualize the ranked lists generated by the statistical debugging step and to let users inspect the blocks (or super

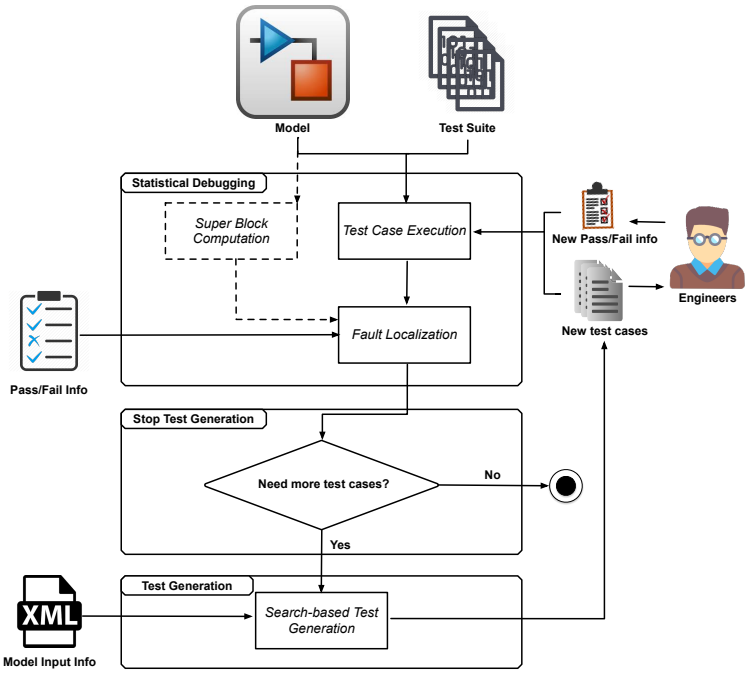


**Fig. 16** Workflow of SimFL

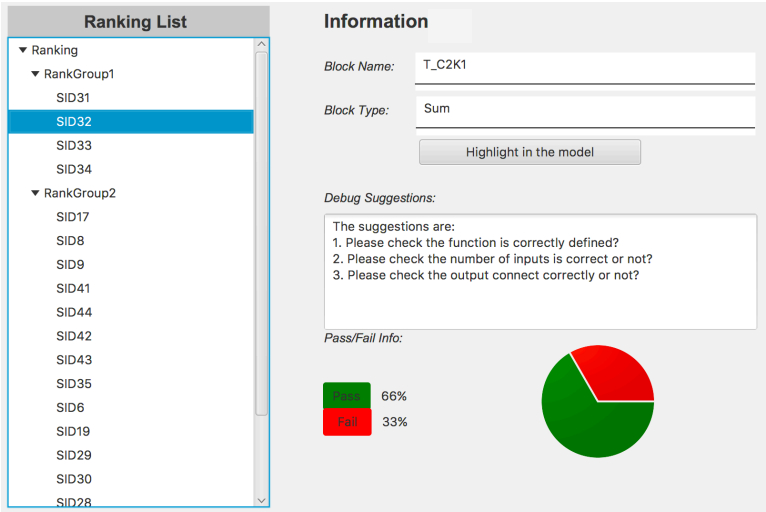


**Fig. 17** The debug window of *SimFL*

**Table 8** The wilcoxon test results ($p$-values) and the $A_{12}$ effect size values comparing distributions in Fig. 15 (b) (i.e., comparing the impact of threshold values on the number of additional test cases generated).

| Pair A vs. B | $p$-value | $A_{12}$ |
|---|---|---|
| THR5 vs. THR10 | 0.00 | 0.58 |
| THR5 vs. THR15 | 0.00 | 0.76 |
| THR5 vs. THR20 | 0.00 | 0.76 |
| THR5 vs. THR25 | 0.00 | 0.83 |
| THR5 vs. THR30 | 0.00 | 0.77 |
| THR5 vs. THR35 | 0.00 | 0.78 |
| THR10 vs. THR15 | 0.00 | 0.68 |
| THR10 vs. THR20 | 0.00 | 0.68 |
| THR10 vs. THR25 | 0.00 | 0.75 |
| THR10 vs. THR30 | 0.00 | 0.71 |
| THR10 vs. THR35 | 0.00 | 0.72 |
| THR15 vs. THR20 | 0.48 | 0.51 |
| THR15 vs. THR25 | 0.00 | 0.60 |
| THR15 vs. THR30 | 0.00 | 0.57 |
| THR15 vs. THR35 | 0.00 | 0.59 |
| THR20 vs. THR25 | 0.00 | 0.60 |
| THR20 vs. THR30 | 0.00 | 0.57 |
| THR20 vs. THR35 | 0.00 | 0.58 |
| THR25 vs. THR30 | 0.00 | 0.48 |
| THR25 vs. THR35 | 0.00 | 0.50 |
| THR30 vs. THR35 | 0.00 | 0.51 |

blocks) in the generated ranked lists. The window in Fig. 17 is divided into two parts: on the left side, *SimFL* presents the generated ranked list in a hierarchical form. At the first hierarchy level, rank groups are shown in the list. Users can expand each rank group to browse and inspect the blocks in each rank group. Recall that the blocks in the same rank group have the same suspiciousness score. When the super block computation module is activated, the ranked list has three hierarchy levels: The first level shows the rank groups, the second level shows the super blocks in each rank group, and the third level shows the atomic blocks inside each super block. The atomic blocks in the rank list are identified based on their unique Simulink ID values (i.e., *Simulink SID*). By clicking on each atomic block in the ranked list, the right side of the window in Fig. 17 is automatically populated with some information about the selected block in the ranked list. In particular, the right part of the window shows the block name, the block type (e.g., constant, arithmetic operation, relation, switch), some bug fixing suggestions and a pie chart illustrating the percentages of pass and failed test cases that have executed the selected block. The bug fixing suggestions are some informal hints describing in what ways a selected block might be faulty. These suggestions depend on the Simulink block type. For example, Fig. 17 shows the bug fixing suggestions related to the "Sum" block. Finally, in order to help users quickly find the exact location of a selected block in the original model, SimFL provides a button ("Highlight in the model" button in Fig. 17) that when it is clicked, the underlying Simulink model is opened and the selected Simulink atomic block is highlighted in the model.

The user interface in Fig. 17 helps users browse the ranked list and inspect the most suspicious blocks in the ranked list. The users may be able to identify the fault

in their model at this step and stop debugging. Otherwise, in the stop test generation step, they need to decide whether rankings are likely to be improved by generating new test cases. This step currently implements the first part of the STOPTESTGENERATION algorithm in Figure 6 (i.e., lines 1 to 4). In particular, it computes the super blocks in the underlying Simulink model, and checks if each of the top ranked groups in the ranked list consists of a single super block. If so, there is no need to proceed to the last step (i.e, the test generation step) since adding new test cases does not improve the statistical ranking results. The *SimFL* tool currently does not implement the part of the STOPTESTGENERATION algorithm that relies on historical data to build prediction models. So in the stop test generation step of *SimFL*, users must decide if they want to move to the last step unless the super block information suggests otherwise.

In the test generation step, *SimFL* uses a search-based test generation algorithm to generate a set number of new test cases. Users can choose the test generation objective among the DBB, Dissimilarity and Density test objectives, and the search algorithm between the HCRR and HC algorithms. They can also specify, based on their test oracle budget, the number of new test cases that they wish to generate. Finally, users should provide the value ranges of Simulink model inputs. Specifically, the value ranges are provided in XML format and used by our test generation algorithms. Once new test cases are generated, users need to provide the oracle information related to the new test cases. They can then reiterate through the steps in Fig. 16 to generate new statistical rankings and to debug their models. They may stop the debugging process when they successfully find a fault in their model or when they run out of budget.

*SimFL* has been implemented as a standalone application in Java. The pre-requisites of the tool are: (i) JDK version later than 1.7, and (ii) Matlab/Simulink version later than 2012. *SimFL* can run in both MacOS and WindowsOS environments. The GUI is implemented in JavaFX. We adopt a third-party Java API Matlab control [31] to invoke execution of the Simulink model in our tool. The tool is available at: `http://sites.google.com/svv.lu/simfl`

## 6 Threats to Validity

In this section we discuss the threats to validity based on the following four perspectives of validity and threats:

**Conclusion validity:** The main threats to conclusion validity arise from not accounting for random variation and inappropriate use of statistics. We mitigate these threats by running each different search algorithm ten times on sixty faulty versions obtained from our three industry models. This led to a large number of observation points to be used as a basis for our comparisons. Following existing guidelines [7], we used non-parametric pairwise Wilcoxon signed-rank test, and Vargha and Delaney's $\hat{A}_{12}$ for our statistical comparisons.

**Internal validity:** In our work, we evaluated our approach on faulty industrial Simulink models where each faulty model contains only one fault. In practice, models may have multiple faults that may impact one another in unknown ways. Hence, our experiment results might be different when our approach is applied to Simulink models

with multiple faults. However, a large bulk of existing research on fault localization is exclusively evaluated on software artifacts (i.e., typically programs) seeded with single faults [11, 17]. To be able to compare our results with those reported in the literature, we decided to be consistent with the existing experiment settings and evaluate our approach on models seeded with single faults.

The input signals used in our experiment are considered to be constant over time (as discussed in Section 2). This is mainly because the subject models used in our experiments are physical plant models that are mostly tested using constant signals in practice. Other Simulink models may be tested using more complex input signals. This however may come at the cost of requiring more time and effort to predict correct signal outputs to determine pass/fail information for each test case.

**Construct validity:** The main threat to construct validity is posed by unsuitable or ill-defined metrics. To this end, we note that we mainly used the following two standard metrics used in the fault localization research to analyze our results: the *absolute number of blocks inspected* to find faults, and the *proportion of faults localized*.

**External validity:** Our approach is focused on fault localization of Simulink models. Simulink is used by more than 60% of the engineers developing Cyber-Physical Systems (CPSs) [83], and is the prevalent modeling language in the automotive domain [79]. In our experiments, we evaluated our approach using three representative industrial subjects from our partner company, Delphi Automotive Systems. We asked a senior Delphi test engineer to provide us with realistic faults for Simulink models. We further ensured that the faults are of different types and are seeded into different parts of the models. We have listed the faults used in our experiments in Section 4.2. Further, the faults used in our work represent the most common faults observed in practice and also used in the existing literature on mutation operators for Simulink models [16, 24, 28, 78, 81]. However, the three Simulink models used in this study were used to model physical plants. It is yet to be seen if our results generalize to Simulink models to capture other aspect of CPSs and Simulink models from other domains, e.g., avionics.

Our approach is designed for situations where test oracles are developed manually. Based on our observations, this is a common situation in many companies in the automotive domain. Other test generation methods [8, 63] can be used in conjunction with fault localization if test oracles can be automated.

To build prediction models utilized in the STOPTESTGENERATION subroutine, our approach requires historical data containing statistical rankings and the position of faulty blocks in those rankings. Such data may not always be available. In this case, the STOPTESTGENERATION subroutine (lines 4 and 5) can be skipped in our approach shown in Fig. 3.

## 7 Related Work

As noted earlier, this article is an extension of a previous conference paper [43]. In this section, we discuss and compare with several other strands of related work in the areas of automated test generation, model debugging and predicting fault localization results.

### 7.1 Automated test generation

Several techniques have been proposed to generate test cases for Simulink models. Some of these techniques focus on generating test suites with high mutant-killing capabilities [16, 29, 81, 82]. A mutant is killed by a test case if the output yields deviations for some output when applied to both the mutant model and the original model. These mutant-killing techniques assess the quality of test suites by measuring the number/percentage of mutants that can be eliminated/identified by a given test suite. Mutation-based techniques can be implemented either using search techniques [81, 82] or behavioral analysis techniques [16, 29].

Other test generation techniques focus on generating coverage-adequate test suites for Simulink models. For example, search-based techniques have been applied to minimize a fitness function that approximates how far a given test input is from covering a specific Simulink block or Stateflow state [73, 74, 80]. Reachability analysis is used to generate coverage-adequate test inputs by measuring unreachability of the faulty model parts [27, 55, 65]. Recently some techniques generate test suites for Simulink models aiming to maximize diversity of the generated test outputs [53, 54].

Almost all test generation approaches discussed above aim to generate test suites with high fault-revealing ability. In other words, the generated test suites are assessed based on their ability to produce outputs differing from the expected results, hence revealing failures. In our work, however, we assume some failures are already revealed using an existing test suite. Our aim is to extend the test suite with new test cases to improve the accuracy of statistical debugging.

A number of test generation techniques specifically focus on generating test cases to improve fault localization [8, 11, 32, 63]. All these techniques are targeted at and evaluated on software code, and none have been applied to Simulink models. Among these techniques, some [8, 63] rely on the presence of an automated test oracle to generate test cases. Recall that one important requirement in our work is that the pass/fail information for each candidate test input is not readily available. Hence, test generation techniques that require such information to improve fault localization [8, 63] are not applicable since these techniques are feasible only when test oracles are automatable.

The test generation techniques proposed by Baudry et. al. [11] and Campos et. al. [17] do not rely on the presence of test oracle information to generate new test cases for fault localization improvement. Both of these techniques attempt to generate test cases that execute varying subsets of program statements. In particular, Baudry et. al. [11] guide test generation by maximizing the number of Dynamic Basic Blocks (i.e., program elements that are always executed together), and Campos et. al. [17] attempt to generate test cases that yield an optimized value for their proposed notion of coverage density. 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 [32] and use it to improve fault localization for Simulink models. In contrast to the above-mentioned work [11, 17, 32], 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.

Very recently, Perez et al. [59] proposed a new test objective referred to as Density-Diversity-Uniqueness (DDU) that combines the DBB [11] and Density [17] test objectives discussed in this paper as well as a third test objective defined based on coverage diversity [35]. The DDU test objective aims to mitigate the limitations of these individual test objectives to generate test cases that lead to improved statistical fault localization results. Given that DDU does not require automated test oracles, we can consider it as an alternative test objective in our work and assess its performance on improving fault localization in Simulink models. We leave to future work integration of DDU into our framework and comparing it with the test objectives studied in this paper.

### 7.2 Model Debugging

There is a growing body of research on techniques to enable model debugging, most particularly in the model-driven engineering community [9]. Some of these techniques focus on providing tools and features to help with manual debugging of software models (e.g. [9,56]). Automated approaches to fault localization and debugging at the level of software models can be broadly divided into the following two categories [9]: First, techniques that enable model debugging at the level of simulation models and rely on simulators or model interpreters. Such techniques complement design-time simulation by helping identify and localize faults in early and executable design models. Second, techniques that translate models into another language, typically a programing language, and largely rely on debuggers or analysis tools available for that language.

Our previous work [42] as well as our current work is focused on applying and improving statistical debugging at the level of Simulink models and fall in the first category described above. To the best of our knowledge our approach is the first to apply statistical debugging at the level of Simulink models to enable early stage and design time analysis of these models.

A number of testing techniques for Simulink models translate Simulink models into code and use existing code analysis tools to detect faults in the models [10, 58]. While these techniques are useful for testing late-stage Simulink models (i.e., code-generation Simulink models [54]), they are not typically applicable to early-stage Simulink models with continuous behavior (e.g., those containing plant models and continuous controllers) [54]. Further, the testing results have to be translated back to the level of Simulink models to be useful for any debugging task. None of the above approaches provide a mechanism to translate the testing results obtained at the code level into interpretable data at the level of the original Simulink models.

Our work relates to the work of Schneider [66] that proposes a technique for tracking the root causes of defects in Simulink. In that technique, engineers identify failures, typically run-time failures, at the level of code generated from Simulink models. The program statement that exhibits the failure is then mapped to a Simulink block, and all the paths leading to that block are collected and assigned weights based

on some heuristic. The path with the highest weight is then reported to the engineer as the root cause of the defects. This work focuses on runtime failures (e.g., division by zero), while in our work, we consider a wider range of fault types for Simulink models (see Section 4.2). Further, Schneider [66] does not provide any realistic evaluation of the proposed approach. Also, as they do not report the number of blocks that engineers need to eventually inspect, the expected debugging effort is not clear either. Finally, Schneider [66] does not propose any mechanism to improve fault localization accuracy when a debugging task turns out to be inconclusive for a given test suite.

### 7.3 Slicing and Analyzing Simulink models

In our previous work [42], we defined test execution slices for Simulink models to support our statistical debugging approach. As noted there, our notion of test execution slice for Simulink models differs from the notion of *execution slice* defined by Agrawal [2] for programs. Specifically, an execution slice is the set of basic blocks or decisions that are executed by a test case to produce *all outputs* in programs [2]. However, test execution slices in our paper are defined per test case and per output. In a similar vein, Approximate Dynamic Backward Slicing (ADBS) [47] bears some similarities to our notion of test execution slice. To clarify the differences between these two, we note that our notion of test execution slice accounts for the blocks that are executed by a test case and affect a specific output generated by that test case. However, ADBS contains program statements executed by a test case and appearing in the backward static slice of a specific output, but these statements do not necessarily affect the output generated by that test case. Finally, Reicherdt and Glesner [61] proposed a slicing method for Simulink models where control dependencies are obtained via Simulink *Conditional Execution Contexts* (CECs) and are used to create static slices based on a set of blocks. In our work, we chose to use model execution information to identify control dependencies and compute slices since the static slicing proposed by Reicherdt et al. [61] and Sridhar et al. [68] based on CECs may provide over approximations that may not be sufficiently precise to determine control dependencies. Moreover, the computation for CECs is more expensive than our test execution slices computation.

We note that our notion of test execution slices is primarily meant to enable statistical debugging for Simulink models. Some recent approaches focus on developing variant slicing and model exploration operators for Simulink models [25, 56] as well as tree-like models [26]. Comparing our slicing approach with these recent slicing techniques and utilizing them in the context of statistical debugging is outside the scope of this paper and left for future work.

### 7.4 Predicting Fault Localization Results

In this paper, we use two strategies to predict whether adding new test cases can improve the fault localization accuracy: (1) super block computation, and (2) building prediction models. Here, we discuss each strategy and contrast it with the relevant related work.

### *7.4.1 Super Block Computation*

As we discussed in Section 3.1, the concept of *Dynamic Basic Block* (DBB) proposed by Baudry et al. [11] bears some similarities with our notion of super block. For a given test suite, a DBB is a subset of statements such that any test case in the test suite executes either all or none of these statements. In contrast, a super block is a subset of Simulink atomic blocks such that any *arbitrary* test case executes either all or none of the blocks in a super block. That is, super blocks are not tied to any specific test suite and are computed purely based on static analysis of Simulink models. Due to the definition of super blocks, when a ranked group consists of a super block only, we can conclude that generating more test cases will not lead to partitioning that rank group. Hence, super blocks can be used as heuristics to predict if new test cases can improve the fault localization accuracy. However, DBBs cannot be used for this purpose.

Our notion of super blocks bears some similarities with program basic blocks [3]. However, a program basic block is a maximal *sequence* of instructions executed together, while our notion of super block is a maximal *set* of blocks that are always executed together, and the blocks in a super block do not have to form a sequence. Moreover, super blocks differ from the notion of postdominance used in control flow graphs [4]. In particular, postdominance is an asymmetric notion: A node $d$ in a control flow graph post-dominates a node $n$ if every path from $n$ to the exit node passes through $d$. This implies that any test case executing $n$ will execute $d$ as well, but there might be test cases that execute $d$ but not $n$. Based on our notion of super blocks, however, nodes $d$ and $n$ are in a super block if and only if every test case executes either both or neither of them.

Finally, as discussed in the background section (i.e., Section 2), in order to adapt statistical debugging to Simulink models, we define the notion of test execution slice over Simulink models. However, since the details of this adaptation as well as computation of test execution slices have been discussed in detail in our previous paper [42], in this paper, we do not further discuss our slicing approach and do not compare it with the related work on slicing Simulink models.

### *7.4.2 Building Prediction Models*

Le and Lo [38] 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 [38] as a test generation stopping criterion.

Xia et al. [77] 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. [77] 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.

The predictability and accuracy of fault localization results might also be impacted by the presence of coincidentally correct test cases (CCT), i.e., test cases that execute a fault but are not able to reveal it, and hence, are counted as a passing test case [72]. A recent study by Masri et al. [48] shows the prevalence of CCT in test suites applied to programs for the purpose of testing or statistical debugging. Some recent papers [6, 19] define information theoretic metrics to predict when an error at an intermediate program statement may fail to propagate to final outputs, and hence, potentially lead to CCT. Such metrics might help us improve fault localization accuracy by identifying and excluding test cases that are likely to be CCT when executed on the underlying Simulink model. We leave investigating this line of research to future work.

## 8 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 are very important in contexts where running and analyzing test cases 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 takes a long time. 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. We further show that varying test objectives used to generate the initial test suites does not have a significant impact on the fault localization results obtained based on those test suites. We evaluate and compare the performance of different prediction models built based on seven alternative feature combinations, and identify the best-performing one. Based on our experiment, our best 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 propose and experiment with alternative test objectives that aim to improve statistical debugging for Simulink models. We further plan 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 [60]. 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.

**Acknowledgements** We gratefully acknowledge funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 694277).

## References

1. Abreu, R., Zoeteweij, P., Van Gemund, A.J.: On the accuracy of spectrum-based fault localization. In: Proceedings of Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, pp. 89–98. IEEE (2007)
2. Agrawal, H., Horgan, J.R.: Dynamic program slicing. ACM SIGPLAN Notices **25**(6), 246–256 (1990)
3. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools. Addison-Wesley Reading (2007)
4. Allen, F.E.: Control flow analysis. SIGPLAN Notices **5**(7), 1–19 (1970)
5. Alshahwan, N., Harman, M.: Coverage and fault detection of the output-uniqueness test selection criteria. In: Proceedings of the 23rd International Symposium on Software Testing and Analysis (ISSTA'14), pp. 181–192. ACM (2014)
6. Androutsopoulos, K., Clark, D., Dan, H., Hierons, R.M., Harman, M.: An analysis of the relationship between conditional entropy and failed error propagation in software testing. In: Proceedings of the 36th International Conference on Software Engineering (ICSE'14), pp. 573–583. ACM (2014)
7. Arcuri, A., Briand, L.: A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Software Testing Verification and Reliability **24**(3), 219–250 (2014)
8. Artzi, S., Dolby, J., Tip, F., Pistoia, M.: Directed test generation for effective fault localization. In: Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA'10), pp. 49–60. ACM (2010)
9. Bagherzadeh, M., Hili, N., Dingel, J.: Model-level, platform-independent debugging in the context of the model-driven development of real-time systems. In: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17), pp. 419–430. ACM (2017)
10. Balasubramanian, D., Pasareanu, C.S., Whalen, M.W., Karsai, G., Lowry, M.: Polyglot: modeling and analysis for multiple statechart formalisms. In: Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA'11), pp. 45–55. ACM (2011)
11. Baudry, B., Fleurey, F., Le Traon, Y.: Improving test suites for efficient fault localization. In: Proceedings of the 28th International Conference on Software Engineering (ICSE'06), pp. 82–91. ACM (2006)
12. Ben Abdessalem, R., Nejati, S., Briand, L.C., Stifter, T.: Testing advanced driver assistance systems using multi-objective search and neural networks. In: Proceedings of the 31st International Conference on Automated Software Engineering (ASE'16), pp. 63–74. ACM (2016)
13. Binder, R.V.: Testing object-oriented systems: models, patterns, and tools. Addison-Wesley Professional (2000)
14. Breiman, L., Friedman, J., Stone, C.J., Olshen, R.A.: Classification and regression trees. CRC press (1984)
15. Briand, L.C., Nejati, S., Sabetzadeh, M., Bianculli, D.: Testing the untestable: model testing of complex software-intensive systems. In: Companion Proceedings of the 38th International Conference on Software Engineering, (ICSE'16), pp. 789–792 (2016)
16. Brillout, A., He, N., Mazzucchi, M., Kroening, D., Purandare, M., Rümmer, P., Weissenbacher, G.: Mutation-based test case generation for simulink models. In: International Symposium on Formal Methods for Components and Objects, pp. 208–227. Springer (2009)
17. Campos, J., Abreu, R., Fraser, G., d'Amorim, M.: Entropy-based test generation for improved fault localization. In: Proceedings of the 28th International Conference on Automated Software Engineering (ASE'13), pp. 257–267. IEEE (2013)
18. Chen, T.Y., Leung, H., Mak, I.: Adaptive random testing. In: Proceedings of the 9th Asian Computing Science Conference, pp. 320–329 (2004)

19. Clark, D., Feldt, R., Poulding, S., Yoo, S.: Information transformation: An underpinning theory for software engineering. In: Proceedings of the 37th International Conference on Software Engineering (ICSE'15), vol. 2, pp. 599–602. IEEE (2015)
20. Cleve, H., Zeller, A.: Locating causes of program failures. In: Proceedings of the 27th International Conference on Software Engineering (ICSE'05), pp. 342–351. ACM (2005)
21. Delphi Automotive Luxembourg: http://www.delphi.com/about/locations/luxembourg
22. Fagin, R., Kumar, R., Sivakumar, D.: Comparing top k lists. SIAM Journal on Discrete Mathematics **17**(1), 134–160 (2003)
23. Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Transactions on Software Engineering (TSE) **39**(2), 276–291 (2013)
24. Gay, G., Rajan, A., Staats, M., Whalen, M.W., Heimdahl, M.P.E.: The effect of program and model structure on the effectiveness of MC/DC test adequacy coverage. ACM Transactions on Software Engineering Methodology (TOSEM) **25**(3), 25:1–25:34 (2016)
25. Gerlitz, T., Kowalewski, S.: Flow sensitive slicing for MATLAB/Simulink models. In: Proceedings of the13th Working IEEE/IFIP Conference on Software Architecture, pp. 81–90. IEEE (2016)
26. Gold, N.E., Binkley, D., Harman, M., Islam, S., Krinke, J., Yoo, S.: Generalized observational slicing for tree-represented modelling languages. In: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17), pp. 547–558 (2017)
27. Hamon, G.: Simulink Design Verifier-Applying automated formal methods to Simulink and StateFlow. In: Proceedings of the 3rd Workshop on Automated Formal Methods (2008)
28. Hanh, L.T.M., Binh, N.T.: Mutation operators for Simulink models. In: Proceedings of the 4th International Conference on Knowledge and Systems Engineering, pp. 54–59. IEEE (2012)
29. He, N., Rümmer, P., Kroening, D.: Test-case generation for embedded Simulink via formal concept analysis. In: Proceedings of the 48th Design Automation Conference, pp. 224–229. ACM (2011)
30. Jaccard, P.: Etude comparative de la distribution florale dans une portion des Alpes et du Jura. Impr. Corbaz (1901)
31. Java API for Matlab: https://code.google.com/archive/p/matlabcontrol/
32. Jiang, B., Zhang, Z., Chan, W.K., Tse, T.: Adaptive random test case prioritization. In: Proceedings of the 24th International Conference on Automated Software Engineering (ASE'09), pp. 233–244. IEEE (2009)
33. Jones, J.A., Harrold, M.J.: Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th International Conference on Automated Software Engineering (ASE'05), pp. 273–282. ACM (2005)
34. Jones, J.A., Harrold, M.J., Stasko, J.: Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering (ICSE'02), pp. 467–477. ACM (2002)
35. Jost, L.: Entropy and diversity. Oikos **113**(2), 363–375 (2006)
36. Kendall, M.G.: A new measure of rank correlation. Biometrika **30**, 81–93 (1938)
37. Kendall, M.G.: Rank correlation methods. Griffin (1948)
38. Le, T.D.B., Lo, D.: Will fault localization work for these failures? an automated approach to predict effectiveness of fault localization tools. In: Proceeding of the 29th International Conference on Software Maintenance (ICSM'13), pp. 310–319. IEEE (2013)
39. Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I.: Scalable statistical bug isolation. ACM Special Interest Group on Programming Languages Notices **40**(6), 15–26 (2005)
40. Liu, B.: Experiments Data. https://github.com/Avartar/TCGenForFL/
41. Liu, B., Lucia, Nejati, S., Briand, L., Bruckmann, T.: Localizing multiple faults in Simulink models. In: Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16), pp. 146–156. IEEE (2016)
42. Liu, B., Lucia, Nejati, S., Briand, L., Bruckmann, T.: Simulink fault localization: an iterative statistical debugging approach. Software Testing, Verification and Reliability Journal pp. 431–459 (2016)
43. Liu, B., Lucia, Nejati, S., Briand, L.C.: Improving fault localization for Simulink models using search-based testing and prediction models. In: Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER'17), pp. 359–370. IEEE (2017)
44. Liu, C., Yan, X., Fei, L., Han, J., Midkiff, S.P.: Sober: statistical model-based bug localization. ACM SIGSOFT Software Engineering Notes **30**(5), 286–295 (2005)
45. Lucia, Lo, D., Xia, X.: Fusion fault localizers. In: Proceedings of the 29th International Conference on Automated Software Engineering (ASE'14), pp. 127–138. ACM (2014)

46. Luke, S.: Essentials of Metaheuristics, vol. 113. Lulu Raleigh (2009)
47. Mao, X., Lei, Y., Dai, Z., Qi, Y., Wang, C.: Slice-based statistical fault localization. Journal of Systems and Software **89**, 51–62 (2014)
48. Masri, W., Assi, R.A.: Prevalence of coincidental correctness and mitigation of its impact on fault localization. ACM Transactions On Software Engineering and Methodology (TOSEM) **23**(1), 8:1–8:28 (2014)
49. Mathur, A.P.: Foundations of software testing. Copymat Services (2006)
50. MathWorks: Simulink. `http://www.mathworks.nl/products/simulink/`
51. MathWorks: StateFlow. `http://www.mathworks.nl/products/stateflow/`
52. Matinnejad, R., Nejati, S., Briand, L., Bruckmann, T., Poull, C.: Search-based automated testing of continuous controllers: Framework, tool support, and case studies. Information and Software Technology Journal **57**, 705–722 (2015)
53. Matinnejad, R., Nejati, S., Briand, L.C., Bruckmann, T.: Effective test suites for mixed discrete-continuous StateFlow controllers. In: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15), pp. 84–95. ACM (2015)
54. Matinnejad, R., Nejati, S., Briand, L.C., Bruckmann, T.: Automated test suite generation for time-continuous Simulink models. In: Proceedings of the 38th International Conference on Software Engineering (ICSE'16), pp. 595–606. ACM (2016)
55. Mohalik, S., Gadkari, A.A., Yeolekar, A., Shashidhar, K., Ramesh, S.: Automatic test case generation from Simulink/StateFlow models using model checking. Software Testing, Verification and Reliability Journal **24**(2), 155–180 (2014)
56. Pantelic, V., Postma, S.M., Lawford, M., Korobkine, A., Mackenzie, B., Ong, J., Bender, M.: A toolset for Simulink - improving software engineering practices in development with Simulink. In: Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development (MODELS'15), pp. 50–61. SciTePress (2015)
57. Parnin, C., Orso, A.: Are automated debugging techniques actually helping programmers? In: Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA'11), pp. 199–209. ACM (2011)
58. Pasareanu, C.S., Schumann, J., Mehlitz, P., Lowry, M., Karsai, G., Nine, H., Neema, S.: Model based analysis and test generation for flight software. In: Proceedings of the 3rd International Conference on Space Mission Challenges for Information Technology, pp. 83–90. IEEE (2009)
59. Perez, A., Abreu, R., van Deursen, A.: A test-suite diagnosability metric for spectrum-based fault localization approaches. In: Proceedings of the 39th International Conference on Software Engineering (ICSE'17), pp. 654–664. IEEE (2017)
60. Rapos, E.J., Cordy, J.R.: Examining the co-evolution relationship between Simulink models and their test cases. In: Proceedings of the 8th International Workshop on Modeling in Software Engineering, pp. 34–40. ACM (2016)
61. Reicherdt, R., Glesner, S.: Slicing MATLAB Simulink models. In: Proceedings of the 34th International Conference on Software Engineering (ICSE'12), pp. 551–561 (2012)
62. Renieris, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: Proceedings of the 18th International Conference on Automated Software Engineering (ASE'03), pp. 30–39. IEEE (2003)
63. Röszler, J., Fraser, G., Zeller, A., Orso, A.: Isolating failure causes through test case generation. In: Proceedings of the 21st International Symposium on Software Testing and Analysis (ISSTA'12), pp. 309–319. ACM (2012)
64. Santelices, R., Jones, J.A., Yu, Y., Harrold, M.J.: Lightweight fault-localization using multiple coverage types. In: Proceedings of the 31st International Conference on Software Engineering (ICSE'09), pp. 56–66. IEEE (2009)
65. Satpathy, M., Yeolekar, A., Peranandam, P., Ramesh, S.: Efficient coverage of parallel and hierarchical StateFlow models for test case generation. Software Testing, Verification and Reliability Journal **22**(7), 457–479 (2012)
66. Schneider, J.: Tracking down root causes of defects in Simulink models. In: Proceedings of the 29th International Conference on Automated Software Engineering (ASE'14), pp. 599–604. ACM (2014)
67. Spearman, C.: The proof and measurement of association between two things. The American journal of psychology **15**(1), 72–101 (1904)
68. Sridhar, A., Srinivasulu, D.: Slicing MATLAB Simulink/StateFlow models. In: Intelligent Computing, Networking, and Informatics, pp. 737–743. Springer (2014)
69. Thums, A., Quante, J.: Reengineering embedded automotive software. In: Proceedings of the 28th International Conference on Software Maintenance (ICSM'12), pp. 493–502. IEEE (2012)

70. Vargha, A., Delaney, H.D.: A critique and improvement of the CL common language effect size statistics of McGraw and Wong. Journal of Educational and Behavioral Statistics **25**(2), 101–132 (2000)
71. Varrette, S., Bouvry, P., Cartiaux, H., Georgatos, F.: Management of an Academic HPC Cluster: The UL Experience. In: Proceedings of the International Conference on High Performance Computing & Simulation, pp. 959–967. IEEE (2014)
72. Wang, X., Cheung, S.C., Chan, W.K., Zhang, Z.: Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In: Proceedings of the 31st International Conference on Software Engineering (ICSE'09), pp. 45–55. IEEE Computer Society (2009)
73. Windisch, A.: Search-based testing of complex Simulink models containing StateFlow diagrams. In: Proceeding of the 31st International Conference on Software Engineering-Companion, pp. 395–398. IEEE (2009)
74. Windisch, A.: Search-based test data generation from StateFlow StateCharts. In: Proceedings of the 12th annual conference on Genetic and evolutionary computation, pp. 1349–1356. ACM (2010)
75. Wong, E., Debroy, V., Gao, R., Li, Y.: The DStar method for effective software fault localization. IEEE Transactions on Reliability **63**(1), 290–308 (2014)
76. Wong, E., Wei, T., Qi, Y., Zhao, L.: A crosstab-based statistical method for effective fault localization. In: Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST'08), pp. 42–51. IEEE (2008)
77. Xia, X., Gong, L., Le, T.D.B., Lo, D., Jiang, L., Zhang, H.: Diversity maximization speedup for localizing faults in single-fault and multi-fault programs. Automated Software Engineering Journal **23**(1), 43–75 (2016)
78. Yin, Y.F., Zhou, Y.B., Wang, Y.R.: Research and improvements on mutation operators for Simulink models. In: Applied Mechanics and Materials, vol. 687, pp. 1389–1393. Trans Tech Publ (2014)
79. Zander, J., Schieferdecker, I., Mosterman, P.J.: Model-based testing for embedded systems. CRC press (2011)
80. Zhan, Y., Clark, J.: Search based automatic test-data generation at an architectural level. In: Proceedings of the 6th Genetic and Evolutionary Computation Conference, pp. 1413–1424. Springer (2004)
81. Zhan, Y., Clark, J.A.: Search-based mutation testing for Simulink models. In: Proceedings of the 7th Genetic and Evolutionary Computation Conference, pp. 1061–1068. ACM (2005)
82. Zhan, Y., Clark, J.A.: A search-based framework for automatic testing of Matlab/Simulink models. Journal of Systems and Software **81**(2), 262–285 (2008)
83. Zheng, X., Julien, C., Kim, M., Khurshid, S.: Perceptions on the state of the art in verification and validation in cyber-physical systems. IEEE Systems Journal **11**, 2614–2627 (2017)