Test Generation Strategies for Building Failure Models and Explaining Spurious Failures

Baharin A. Jodat
balia034@uottawa.ca
University of Ottawa, Canada

Abhishek Chandar
achan260@uottawa.ca
University of Ottawa, Canada

Shiva Nejati
snejati@uottawa.ca
University of Ottawa, Canada

Mehrdad Sabetzadeh
m.sabetzadeh@uottawa.ca
University of Ottawa, Canada

ABSTRACT

Test inputs fail not only due to a fault in the system under test, but also because the inputs are invalid or unrealistic. Such failures are spurious. Testing becomes more effective in exercising the main function of a system if we can avoid spurious failures. In order to identify spurious failures, we propose to build failure models that can explain when a failure occurs and when it does not. In order to generate test inputs from which failure models can be inferred, we examine two alternative approaches: surrogate-assisted and ML-guided test generation. The surrogate-assisted test generation relies on ML to predict labels for test inputs instead of executing them all, while ML-guided test generation aims to infer boundary regions separating passing and failing test inputs and samples test inputs from those regions. We evaluate the accuracy of failure models inferred based on the data generated by these two approaches. Using case studies from the CPS and network domains, we show that our proposed dynamic surrogate-assisted approach generates failure models with an average accuracy of 83%, significantly outperforming the ML-guided and random baselines. In addition, we obtain conditions identifying spurious failures from failure models for two of our case studies and validate these conditions based on expert feedback.

ACM Reference Format:
Baharin A. Jodat, Abhishek Chandar, Shiva Nejati, and Mehrdad Sabetzadeh. 2023. Test Generation Strategies for Building Failure Models and Explaining Spurious Failures. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Traditionally, software testing has been concerned with finding inputs that reveal failures in the system under test (SUT). However, failures do not always indicate faults in the SUT. Instead, failures may arise because the test inputs are *invalid* or *unrealistic*. For example, in an airplane autopilot system, a failure indicating that the ascent requirement fails for a test input that points the plane nose downward would be invalid. This is because the failure is caused by an unmet pre-condition: the nose should be upward for ascent. For another example, consider a network-management system. In such a system, quality-of-service requirements will inevitably fail for unrealistic test inputs that overwhelm the system beyond its capacity. We refer to failures arising from invalid or unrealistic test inputs as *spurious failures*.

Automated random testing (fuzzing) [38] becomes more effective in exercising the main functions of a system if the fuzzer avoids spurious failures [11, 20, 21]. Spurious failures particularly pose a challenge for compute-intensive (CI) systems, where a single test execution takes significant time to complete. For CI systems, we want to use the limited testing time budget to generate valid inputs that exercise the system's main functions. A promising approach for identifying spurious failures is to build *failure models* [20, 27, 40]. Failure models provide conditions that explain the circumstances of failures and describe when a failure occurs and when it does not [27]. Failure models can infer rules leading to and only to failures. These rules are candidates to be validated against domain knowledge to determine whether the failures that the rules identify are spurious.

Recent research on synthesizing input grammars [8, 9, 29, 30, 47] and abstracting failure-inducing inputs [11, 20, 21, 27, 28] aims to understand the circumstances of different failures. These approaches start from an example failure and iteratively generate more tests to learn the input conditions that lead to that failure. The tests are generated via fuzz testing with or without an input grammar. These approaches are geared towards systems with string inputs, where oracles are typically binary (pass/fail) verdicts. However, these approaches are not optimized for systems with numeric inputs, where the inputs are not governed by grammars and where quantitative fitness functions, developed based on system requirements, are used to determine the degree to which test inputs pass or fail. These quantitative fitness functions allow us to use several test-generation heuristics and learning algorithms to explore the input space and generate sets of tests that have enough information to support the inference of candidate rules for spurious failures.

This paper proposes a framework to infer failure models for compute-intensive (CI) systems with numeric inputs. Examples of such systems include cyber-physical systems (CPS) and network systems. We follow a data-driven approach and infer failure models by harvesting information from a set of test inputs. To generate such sets, one can use either *explorative* or *exploitative* search methods [33]. The former attempts to sample the entire search space, whereas the latter attempts to sample the most informative regions

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

of the search space. The challenge with the explorative approach is that we need to collect and execute many test inputs from the search space to determine if they pass or fail. For CI systems, this takes significant time and may become infeasible. The challenge with the exploitative approach is that one needs effective guidance for sampling within large and multi-dimensional search spaces.

Machine learning (ML) has been used for improving the effectiveness and efficiency of both explorative and exploitative search [10, 19, 25, 31, 34, 36, 44]. For explorative search, *surrogate-assisted test generation* relies on ML to predict verdicts for test inputs instead of executing them all [10, 22, 24, 34]. Using a quantitative surrogate, one can forego system executions when the predicted verdicts remain valid after offsetting prediction errors. Otherwise, we execute the SUT and use the results from the executed test inputs to refine the surrogate. As for exploitative search, *ML-guided test generation* aims to infer boundary regions that separate passing and failing test inputs and to subsequently sample test inputs from those regions [19, 31]. The intuition is that tests sampled in the boundary regions are more informative for identifying failures and can further be used to refine the boundary regions. Both approaches provide a set of labelled test inputs from which we infer failure models using decision-rule learning [48]. Ultimately, human experts need to review the rules produced by failure models and determine whether they represent genuine spuriousness. Decision-rule models are an interpretable and efficient way to express failure models as they generate concise and understandable rules that explicitly relate to the system inputs. This makes them suitable for human interpretation and validation.

Our evaluation aims to answer two main questions: (1) How accurate are the failure models generated by the surrogate-assisted and ML-guided techniques in predicting failures? (2) How useful are failure models for identifying spurious failures? We use two kinds of study subjects in our evaluation: (i) A benchmark of four CPS Simulink subjects with 12 requirements that are non-compute intensive (non-CI). This benchmark is used to identify the optimal surrogate-assisted technique and to answer the first question. (ii) Two industrial CI systems, one from the CPS and the other from the network domain. These are used to answer both questions. In summary, we make the following contributions:

(1) We propose a dynamic surrogate-assisted algorithm that uses multiple surrogate models simultaneously during search, and dynamically selects the prediction from the most accurate model (Section 3.2). Our evaluation performed based on seven surrogate model types in the literature [16, 17, 22, 44] shows that, compared to using surrogate models individually, our dynamic surrogate-assisted algorithm provides the best trade-off between the accuracy and the efficiency by generating datasets that are at least 33% larger while being at least 28% more accurate (RQ1 in Section 4).

(2) We compare the accuracy of failure models obtained using our dynamic surrogate-assisted approach against two ML-guided techniques as well as a random search baseline. Our results show that our dynamic surrogate-assisted algorithm yields failure models with an average accuracy, precision, and recall of 86%, 72%, and 88%, respectively, significantly outperforming the two ML-guided algorithms and the random baseline (RQ2 in Section 4).

(3) We demonstrate that, for our CI subjects, the failure models built based on our dynamic surrogate-assisted algorithm provide interpretable failure-inducing rules that lead to spurious failures. For both systems, we validate the inferred rules against domain knowledge to determine whether the resulting failures are genuinely spurious (RQ3 in Section 4).

Organization. Section 2 motivates our approach. Section 3 describes our framework for generating failure models. Section 4 evaluates our approach. Section **??** compares with the related work. Section **??** concludes the paper.

2 MOTIVATION

Using two real-world, compute-intensive (CI) systems, we motivate the need for identifying spurious failures. These systems, both of which are open-source, are a Network Traffic Shaping System (NTSS) [23, 26] and an autopilot system [1].

NTSS is typically deployed on routers to ensure high network performance for real-time streaming applications such as teleconferencing (e.g., Zoom). Without NTSS, voice and video packets may be transmitted out of sequence or with delays. As a result, users may experience choppy or freezing voice/video. Due to the increasing remote-working practices, multiple streaming applications may be running at the same time in homes and small-office settings. This has made systematic testing of NTSS essential as a way to ensure that networks meet their quality-of-experience requirements.

NTSS works by dividing the total network bandwidth into classes with different priorities. The higher-priority classes are typically used for transmitting time-sensitive, streaming voice and video to minimize interruptions. To test the performance of an NTSS, we assign data flows with different bandwidth values to different NTSS classes. The purpose is to ensure that NTSS is configured optimally and can maintain good performance even when a high volume of traffic flows through its different classes. When we stress test an NTSS, no matter how well-designed the NTSS is, we expect the quality of experience to deteriorate and become unacceptable eventually. Test inputs that stress NTSS beyond a certain limit deterministically fail and do not help reveal flaws or suboptimality in the NTSS design. Our approach in this paper infers the limit on the traffic that can flow through different NTSS classes without compromising the quality of experience. As we discuss in Section 4, for an NTSS setup with eight classes from class0 to class7, we learn the following rule, specifying spurious failing test inputs:

r1: IF (class5+ class6+ class7 > 0.75 · threshold) THEN FAIL, where threshold is the sum of the maximum bandwidths of classes 5, 6, and 7. Rule r1 indicates that attempting to simultaneously utilize classes 5, 6 and 7 more than 75% of their maximum ranges would compromise quality of experience for the entire network. While r1 is not documented, it closely matches domain experts' intuition: These three classes have the highest priority, and, because of the NTSS design, high simultaneous utilization of these three classes would starve the lower-priority classes, leading to an unacceptable overall quality of experience. Rule r1 helps domain experts in at least two ways: (1) it informs them that test inputs that satisfy the rule are spurious, since such test cases do not reveal design faults, and (2) it provides experts with data-driven evidence that the cumulative utilization of classes 5, 6, and 7 should be kept below the identified limit.

Our second case study is an autopilot system from the Lockheed Martin benchmark of challenge Simulink models [12]. This system

is expected to satisfy the following requirement: φ= "*When the autopilot is enabled, the aircraft should reach the desired altitude within 500 seconds in calm air*". When we test the autopilot by fuzzing, we find several test inputs that violate this requirement and several test inputs that satisfy it. It is however unclear whether the failures are due to faults in the system or due to missing or unknown assumptions on the system inputs. Any failures caused by missing or unknown assumptions would be spurious.

As we discuss in Section 4, we identify the following rule as one that indicates spurious failures:

r2: IF (PitchWheel[0..300] $\leq$ -28 $\wedge$ Throttle[0..300] $\leq$ 0.1) THEN FAIL

Here, Throttle[0..300] is the boost applied to the engine by the pilot during the first 300s, and PitchWheel[0..300] is the upward or downward degree of the aircraft nose, again during the first 300s. Note that both Throttle and PitchWheel are signals over time. In order to validate r2, we examined the handbook of the De Havilland Beaver aircraft [7]. According to the handbook, for this aircraft type, to satisfy requirement φ, the pilot should manually adjust the throttle boost (Throttle) to a sufficiently high value. The handbook further states that to be able to ascend, the plane's nose should not be pointing downward. That is, r2 describes a situation where the pre-conditions for φ are not met. Hence, the tests in these ranges are expected to fail and uninteresting for revealing system faults. Further, r2 can be used for implementing safeguards against misuse by the human operator (pilot).

3 GENERATING FAILURE MODELS

Figure 1 shows our framework for generating failure models. The inputs to our framework are: (1) an executable system or simulator S, (2) the input-space representation $\mathcal{R}$ for S, and (3) a quantitative fitness function F for each requirement of S; an example requirement, φ, for autopilot was given in Section 2. The full set of requirements for our case studies of Section 4 is available online [2]. We make the following assumptions about the search input space ($\mathcal{R}$) and the fitness function (F): (**A1**) We assume that the system inputs are variables of type real or enumerate. For each real variable, the range of the values that the variable can take is bounded by an upper bound and a lower bound. (**A2**) For each requirement of S, we have a fitness function F based on which a pass/fail verdict can be derived for any test input. Further, the value of F differentiates among the pass test inputs, those that are more acceptable, and among the fail test inputs, those that trigger more severe failures. Specifically, we assume that the range of F is an interval $[-a, b]$ of $\mathbb{R}$. For a given test, $F(t) \geq 0$ iff t is passing, and otherwise, t is failing. The closer $F(t)$ is to b, the higher the confidence that t passes; and the closer $F(t)$ is to $-a$, the higher the confidence that t fails. Assumptions **A1** and **A2** are common for CPS models expressed in Simulink [6, 12, 37, 45], automated driving systems [22], and network-management systems [26], and are valid for all the case studies we use in our evaluation (see Section 4)

As discussed in Section 1, we examine two alternative test-generation approaches for building failure models: *surrogate-assisted* and *ML-guided*. Both approaches can be captured using the framework in Figure 1: The preprocessing phase generates a set of test inputs labelled with fitness values. The main loop takes the test-input set created by the preprocessing phase, and trains a model.

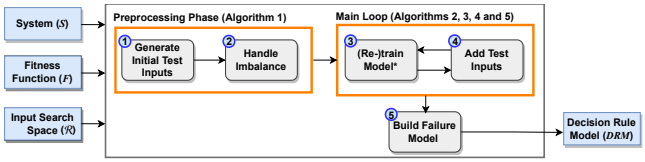


Figure 1: Our framework for generating failure models. The framework captures two test-generation strategies: Surrogate-assisted and ML-guided. For surrogate-assisted test generation, we use Algorithms 2 or 3 to realize the main loop. For ML-guided test generation, we use Algorithms 4 and 5 to realize the main loop.

```
1  Algorithm 1: Preprocessing Phase (InitializeAndBalance)
2  Input S: System
3  Input R = {R_1, ..., R_n}: Ranges for input variables v_1 to v_n
4  Input F: Fitness Function
5  Input d: The budgeted dataset size
6  Output DS: A set of test inputs and their fitness values
7
8  DS^i ← TestGeneration(R, d/2); // (Adaptive) Random Testing
9  DS^I ← Execute(S, DS^i, F); // Compute fitness values
10 DS^b ← HandleImbalance(DS^I);
11 DS^{b,I} ← Execute(S, DS^b, F); // Compute fitness values
12 DS ← DS^{b,I} ∪ DS^I;
13 return DS
```

Figure 2: Algorithm 1: Preprocessing Phase

When the framework is instantiated for surrogate-assisted test generation, the model predicts fitness values for the generated test inputs. When the framework is instantiated for ML-guided test generation, the model guides test-input sampling. The main loop extends the test-input set using the trained model while also refining the model based on newly generated tests. After the main loop terminates, the framework uses the test-input set to train, using decision-rule learners, a failure model. In the remainder of this section, we detail each step of the framework in Figure 1.

3.1 Preprocessing Phase

Algorithm 1 in Figure 2 describes the preprocessing phase that generates the initial dataset for training a model to be used in the main loop of Figure 1. This algorithm first randomly generates half ($d/2$) of the budgeted test inputs and computes a fitness value for each test input by executing the test input using S (lines 8-9). Since ML models perform poorly when the training set is imbalanced, we attempt to address any potential imbalance before using the data for ML training [48] (line 10). In our work, the imbalance, if one exists, is between the pass and the fail classes. We use the well-known synthetic minority over-sampling technique (SMOTE) for addressing imbalance [13]. Let *minor* (resp. *major*) be the number of tests in the minority (resp. majority) class. SMOTE over-samples the minority class by taking each minority-class sample and introducing synthetic examples along the line segments joining any/all of the k minority class nearest neighbours [13]. The process is repeated until we have $m = major - minor$ new such tests.

We discard the labels from SMOTE and instead execute the tests to compute their actual fitness values (line 11). The final dataset (DS) is returned at the end (line 13). Although not shown in Algorithm 1,

```
1  Algorithm 2: Surrogate-assisted test generation
2  Input S: System
3  Input R = {R_1, ..., R_n}: Ranges for input variables v_1 to v_n
4  Input F: Fitness Function
5  Input Budget: Maximum number of fitness evaluations
6  Output DS: A data set to train failure models
7
8  DS ← InitializeAndBalance(S, R, F, Budget/2);
9  DS^I ← DS; flag ← ⊤;
10 while Budget/2 do
11 |  if (flag)
12 |  |  (SM, e) ← Train(DS^I); // Training SM; e is the error
13 |  end
14 |  t ← TestGeneration(R, 1);
15 |  F̄(t) ← SM.Predict(t); // Predict the fitness for t
16 |  if (∃ ∼∈ {≥, <} · (F̄(t) ∼ 0) ∧ (F̄(t) ± e ∼ 0)
17 |  |  DS ← DS ∪ {⟨t, F̄(t)⟩};
18 |  |  flag ← ⊥; // Predicted fitness is used instead of the actual
   |  |       fitness
19 |  else
20 |  |  ⟨t, F(t)⟩} ← Execute(S, {t}, F); // Compute fitness
21 |  |  DS^I ← DS^I ∪ {⟨t, F(t)⟩}; DS ← DS ∪ {⟨t, F(t)⟩}; flag ← ⊤;
22 |  end
23 end
24 return DS
```

Figure 3: Test Generation using Surrogates

to have exactly d tests in DS, we generate the remaining $(d/2 - m)$ tests randomly. Generating these remaining tests randomly does not introduce a new imbalance problem because if random test generation leads to major imbalance, then m is already close to $d/2$ and only a few additional tests may need to be generated. Otherwise, a small m indicates that random testing is relatively balanced; in that case, no special provision is necessary for imbalance mitigation in the randomly generated datatset. Since we discard the labels generated by SMOTE and compute the actual labels, the imbalance problem may in principle persist even after applying SMOTE. For our experiments in Section 4, most synthetic samples generated by SMOTE indeed turn out to belong to the minority class. Our preprocessing therefore successfully addresses imbalance in our case studies.

3.2 Main Loop

The main data-generation loop is realized via two alternative algorithms, described below: surrogate-assisted and ML-guided.

Surrogate-Assisted Test Generation. Algorithm 2 in Figure 3 describes surrogate-assisted test generation. Using surrogates, we predict fitness values for some test inputs and thus do not execute the system for all test inputs. Hence, surrogates help explore a larger portion of the input space and generate larger test sets.

Algorithm 2 uses the dataset, DS, obtained from Algorithm 1 (line 8), to train a surrogate model SM and to compute its error (line 12). We train SM using 80% of DS and compute the mean absolute error of SM on the remaining 20% of DS. The split ratio is based on the well-known 80/20 rule [41]. Next, we randomly generate a test input t and use SM to obtain its predicted fitness value $\bar{F}(t)$ (lines 14–15). We then determine if shifting $\bar{F}(t)$ by the prediction error e of SM would change the verdict for t from pass to fail or vice versa (line 16). If shifting $\bar{F}(t)$ by the error does not impact the test verdict, we deem $\bar{F}(t)$ to be accurate enough to determine the label of t in the final dataset, and hence, we do not execute S for t. In this case, t along with its predicted fitness value is added to DS (line 17). Otherwise, we execute S to compute the actual fitness value for t and add t along with its actual fitness value to DS (lines 20–22). In the latter case (i.e., lines 20–22), we re-train the surrogate model SM using the dataset that includes the new test input and its actual fitness value (lines 11–12). The algorithm returns the final dataset when the time budget runs out (line 25).

Table 1: Surrogate Models and their descriptions.

Name	Description	Name	Description
GPR-L	Gaussian Process Regression – nonparametric Bayesian with linear kernel.	RT	regression tree.
GPR-NL	Gaussian Process Regression – nonparametric Bayesian with nonlinear kernel.	RF	random forest.
LSB	Gradient Boosting – an ensemble of regression trees.	SVR	Support Vector Regression
NN	a two-layer feedforward Neural Network.		

```
10 Algorithm 3: Dynamically Selecting Surrogates
11 ...
12 for i = 0 to sm do
13 |  (SM_i, e_i) ← Train(DS^I); // Train several surrogate models
14 end
15 (SM, e) ← Select SM ∈ {SM_1, ..., SM_sm} with the lowest error e
16 ...
```

Figure 4: Dynamically Selecting Surrogates

In our experimentation (Section 4), we consider the surrogate-model types shown in Table 1. These surrogate-model types are the most widely used ones in the evolutionary search and software testing literature [16, 17, 22, 44]. As suggested by the literature and also as we show in our evaluation (Section 4), no surrogate-model type consistently outperforms the others [18, 49]. Therefore, it is recommended to use a combination of surrogate models. In this paper, we propose, to our knowledge, a novel variation of Algorithm 2 where we train multiple surrogate models and use for predicting fitness values the model that has the lowest error. This variation is shown in Algorithm 3 in Figure 4 where we change line 12 of Algorithm 2 to train and tune a list of surrogate models instead of just one model. We then select the surrogate model with the lowest error for making predictions until the next time we re-train the models. Similarly, each time we execute line 12 of Algorithm 2, we re-train a list of surrogate models and select the one that has the lowest error. We refer to our proposed variation as *dynamic* surrogate-assisted test generation.

In both Algorithm 2 and the variation suggested in Algorithm 3, the first time we train a surrogate model, we also tune its hyperparameters using Bayesian optimization [42]. We use the same tuned hyperparameters in all future iterations. The cost of training and tuning surrogate models for the first time is on the same scale as the cost of a single execution of our CI systems. The time for subsequent re-training of surrogate models is nonetheless negligible since re-training does not involve any tuning. As we discuss in Section 4, the overhead of re-training surrogate models does not deteriorate performance compared to other alternatives.

ML-Guided Test Generation. ML-guided test generation uses ML models for identifying the boundary regions that discriminate pass and fail test inputs and iteratively concentrating test-input sampling to those regions. The idea is that, irrespective of the separability of the set of test inputs, ML models can shift the focus of sampling from the homogeneous regions where either fail or pass verdicts are scarce to regions where neither fail nor pass would be dominant. We consider two alternative ML models that can help us sample from

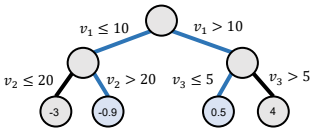
```
1  Algorithm 4: ML-guided test generation (Regression Tree)
2  Input S: System
3  Input ℛ = {R_1, ..., R_n}: Ranges for input variables v_1 to v_n
4  Input F: Fitness Function
5  Input Budget: Maximum number of fitness evaluations
6  Output DS: A data set to train failure models
7
8  DS ← InitializeAndBalance(S, ℛ, F, Budget/2);
9  while Budget/2 do
10 |  RegTree ← Train(DS);
11 |  {R'_{i_1}, ..., R'_{i_m}} ← ExtractBoundaryRanges(RegTree);
12 |  for j ← 1 to m do
13 |  |  ℛ ← (ℛ \ {R_{i_j}}) ∪ {R'_{i_j}}; // Replace ranges in ℛ with those
   |  |      obtained from the regression tree
14 |  end
15 |  {t} ← TestGeneration(ℛ, 1);
16 |  DS ← DS ∪ Execute(S, {t}, F); // Compute fitness for t and
   |      add to DS
17 end
18 return DS
```

Figure 5: ML-guided test generation (Regression tree)

such boundary regions: regression trees (Algorithm 4 in Figure 5) and logistic regression (Algorithm 5 in Figure 6). As we describe below, a regression tree approximates pass-fail boundaries in terms of predicates over inputs variables, while logistic regression infers a linear formula over input variables.

Algorithm 4 uses the *DS* dataset obtained from Algorithm 1 (the preprocessing phase) to train a regression-tree model (lines 8–10). In our regression-tree models, tree edges are labelled with predicates $v_i \sim c$ such that v_i is an input variable, $c \in \mathbb{R}$ is a constant and $\sim \in \{\leq, >\}$. The tree leaves partition the given dataset into subsets such that information gain is maximized [48]. Each leaf is labelled with the average of the fitness measures of the test inputs in that leaf. Provided with a regression tree, Algorithm 4 identifies predicates $\{v_{i_1} \sim c_{i_1}, \ldots, v_{i_m} \sim c_{i_m}\}$ that appear on the two paths whose leaf-node values are closest to zero (one above and one below zero). These predicates specify the boundary between pass and fail, and, as such, we call them boundary predicates. By simplifying the boundary predicates, each variable can have at most one upper-bound predicate ($v \leq c$) and at most one lower-bound predicate ($v > c$). For each predicate $v_{i_j} \sim c$ where $\sim \in \{\leq, >\}$, the algorithm replaces the existing range R_{i_j} of v_{i_j} with $R'_{i_j} = [c-5\% \cdot c, c+5\% \cdot c]$ (lines 12–14). This will ensure that we sample v_{i_j} within the 5% margin around the constant c. The variables that do not appear in the boundary predicates retain their range from the previous iteration. Next, the algorithm generates a test input within the constrained search space (line 15), executes the test input, and adds it along with its fitness measure to *DS* (line 16). The algorithm returns the final dataset after the time budget runs out (line 18).

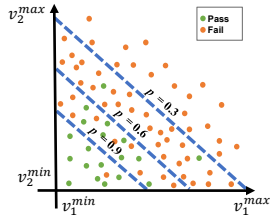
To illustrate range reduction for variables using regression trees, consider the example on the right. We choose the two thicker paths highlighted in blue since their leaf-node values are closest to zero. The new reduced ranges for v_1, v_2 and v_3 are [9.5, 10.5], [19, 21] and [4.75, 5.25] respectively. By sampling within these ranges, we get to focus test-input generation on the pass-fail boarder identified by the regression tree.




```
1  Algorithm 5: ML-guided test generation (Logistic Regression)
2  Input S: System
3  Input ℛ = {R_1, ..., R_n}: Ranges for input variables v_1 to v_n
4  Input F: Fitness Function
5  Input Budget: Maximum number of fitness evaluations
6  Output DS: A data set to train failure models
7
8  DS ← InitializeAndBalance(S, ℛ, F, Budget/2);
9  while Budget/2 do
10 |  LogReg ← Train(DS);
11 |  p ← Probability(DS); // probability of pass in DS
12 |  t ← GenerateCloseToRegBorder(ℛ, LogReg, p); // select a
   |      test close to the regression border for p
13 |  DS ← DS ∪ Execute(S, {t}, F);
14 end
15 return DS
```

Figure 6: ML-guided test generation (Logistic Regression)

Similar to Algorithm 4, Algorithm 5 uses the *DS* dataset from the preprocessing phase to train a logistic regression model (lines 8–10). Since logistic regression is a classification technique, the quantitative labels in *DS* are replaced with pass/fail labels before training. A linear logistic regression model is represented as $\log(\frac{p}{1-p}) = c + \sum_{i=1}^{n} c_i v_i$ where $v_1, \ldots, v_n$ are the input variables, c_i's and c are co-efficients, and p is the probability of the pass class [4, 48]. The algorithm then randomly samples a few test inputs in the search space and picks the one closest to the logistic regression formula obtained by setting p to the percentage of the pass labels in *DS* (line 11). This formula approximates the region that includes a mix of pass and fail test inputs as shown in the figure to the right. The figure shows logistic regression lines with two variables and different values for p. The line with $p = 0.9$ identifies a region where the majority of test inputs pass. On the other end of the spectrum, the line with $p = 0.3$ identifies a region where the majority of test inputs fail. Setting p to the percentage of pass in *DS* is a heuristic to identify a region that includes a mix of pass and fail test inputs. The test input selected in line 12 along with its fitness measure computed using S is added to *DS* (line 13). The algorithm re-trains the logistic regression model whenever a test input has been added to *DS* (line 10). The algorithm returns the final dataset when the time budget runs out (line 15).



Similar to Algorithm 2, the hyperparameters of the regression-tree and logistic-regression models are tuned using Bayesian optimization the first time the models are built, and the same tuned hyperparameters are used in all future iterations.

3.3 Building Failure Models

The output of the main loop in Figure 1 is a set *DS* of tuples $\langle t, F(t) \rangle$ where t is a test input and $F(t)$ is its fitness value. We first convert *DS* into a dataset where test inputs are labelled by *pass* and *fail* labels. Provided with a labelled dataset, we use decision-rule models built using RIPPER [14] to train failure models. Decision-rule models generate a set of IF-condition-THEN-prediction rules where the condition is a conjunction of predicates over the input features and the prediction is either pass or fail. In Section 2, we already showed

two examples of such rules for spurious failures. When no domain knowledge is available, one can directly use the input variables of the system (S) as features for learning. When domain knowledge is available, feature design for decision-rule models can be improved in two ways: (1) Excluding input variables that are orthogonal to the requirement under analysis. For example, the prerequisite for the requirement φ in Section 2 is that the autopilot should be enabled, i.e., APEng = on. As far as test generation for φ is concerned, we need to set APEng = on, since otherwise, φ holds vacuously. We thus do not use APEng as an input feature. For another example, in φ, we do not use the desired altitude as an input feature either, since the system is expected to satisfy φ for any desired altitude in the default range. (2) Using domain knowledge to formulate features over multiple input variables. For NTSS, as discussed in Section 2, the goal is to identify limits on the traffic that can flow through NTSS classes without compromising network quality. Based on domain knowledge, we know that flows have a cumulative nature. Hence, for NTSS, we use as features *sums of subsets* of flow variables. Naturally, like in any feature engineering problem, one can hypothesize alternative ways of formulating the features and empirically determine the formulation leading to highest accuracy [39].

4 EVALUATION

In this section, we empirically evaluate our approach by answering the following research questions (RQs):

RQ1 (configuration). *Which surrogate-assisted technique offers the best trade-off between accuracy and efficiency?* We compare *eight* surrogate-assisted algorithms. These eight algorithms are: (a) Algorithm 2 used with the seven surrogate models in Table 1 individually, and (b) the dynamic surrogate algorithm (Algorithm 3) that uses the seven surrogate models simultaneously and selects the best prediction dynamically. To measure accuracy, we check the correctness of the labels of the tests in the generated datasets; and, to measure efficiency, we evaluate the size of the generated datasets. We use the optimal algorithm for answering RQ2 and RQ3.

RQ2 (effectiveness). *How accurate are the failure models generated by the surrogate-assisted and ML-guided techniques?* We evaluate and compare the accuracy of the failure models obtained by surrogate-assisted and ML-guided algorithms as well as those obtained based on randomly generated test inputs (random baseline).

RQ3 (SoTA Comparison). *How accurate are the failure models generated in RQ2 compared to those generated by the state of the art (SoTA)?* We use the top-performing technique from RQ2 to compare against SoTA. Among the existing approaches that build failure-inducing models [20, 27, 28], we select the Alhazen framework [27], since it uses interpretable machine learning. While Alhazen is geared towards systems with structured inputs (as opposed to systems with numeric inputs, i.e., the focus of our work), in the absence of baselines for systems with numeric inputs, Alhazen is our best baseline for comparison. To be able to compare with Alhazen, we adapt it to numeric-input systems.

RQ4 (Usefulness). *How useful are failure models for identifying spurious failures?* We answer this question for the most accurate failure models from RQ2 and for the two CI systems, namely NTSS and autopilot, discussed in Section 2. NTSS and autopilot are representative examples of industrial systems in the network and CPS domains, respectively. For both systems, we validate the failure-inducing rules against domain knowledge to determine whether the resulting failures are genuinely spurious.

Study Subjects. Our study subjects, which are listed in Table 4, originate from the network and CPS domains. Below, we discuss that the test inputs of our study subjects are vectors of numeric variables. Further, for our subjects, we can define fitness functions with the characteristics specified in Section 3.

Network-system subject. Our network-system subject is the Network Traffic Shaping System (NTSS) discussed in Section 2. To test NTSS, we transmit flows with different bandwidth values into different NTSS classes. A test input for NTSS is defined as a tuple $t = (v_1, \ldots, v_n)$ where n is the number of NTSS classes, and each variable v_i represents the bandwidth of the data flow going through class i. The fitness function for NTSS measures the network quality based on the well-known mean opinion score (MOS) metric [43]. This fitness function ensures the characteristics specified at the beginning of Section 3. For our experiments, we use an NTSS setup based on an industrial small-office and home-office use case that is publicly available [26]. This setup runs Common Applications Kept Enhanced (CAKE) [23], which is an advanced and widely used traffic-shaping algorithm. The setup uses the 8-tier mode of CAKE known as diffserv8 [5, 23], i.e., the number of NTSS classes is 8.

Simulink subjects. Simulink [12] is a widely used language for specification and simulation of CPS. The inputs and outputs of a Simulink model are represented using signals. A typical input-signal generator for Simulink characterizes each input signal using a triple (int, R, n) such that int is an interpolation function, R is a value range, and n is a number of control points [6, 45]. Let (x, y) be a control point. The value of x is from the signal time domain, and the value of y is from range R. Provided with n control points, the interpolation function R (e.g., piecewise constant, linear or piecewise cubic) constructs a signal by connecting the control points [45]. It is usually assumed that the input signals for a Simulink model have the same time domain. Further, for the purpose of testing, we make the common assumption that the control points are equally spaced over the time domain, i.e., the control points are positioned at a fixed time distance [6, 45]. Hence, to generate test inputs, we only need to vary the y variable of control points in the range R. Note that the type of the interpolation function is fixed for each Simulink model input as the interpolation function type is determined by the meaning of the input signal. For example, a reference signal is often a constant or step function. As a result, we can exclude the interpolation function from the test-input signals, and define each test-input signal as a vector of n control variables taking their values from R. Simulink models often have clearly stated requirements. To define a fitness function for each requirement, we encode the requirement in RFOL [37] – a variant of the signal temporal logic which is shown to be expressive for capturing CPS requirements. We use the RFOL semantic function, which is quantitative and satisfies the conditions discussed in Section 3, as the fitness function.

For our experiments, we use a public-domain benchmark of Simulink specifications from Lockheed Martin [3]. The benchmark provides representative CPS systems and is shared by Lockheed as a set of verification-and-validation challenge subjects for researchers and quality-assurance tool vendors. The benchmark includes eleven

Table 2: Parameter names, descriptions and values used by surrogate-assisted (SA), ML-guided test generation (LR and RT) as well as random baseline (RS) algorithms for our 16 subjects.

Parameter Name	Description	SA	RS	LR	RT
maxSimNum	Maximum number of fitness evaluations for a given run for all Simulink models excluding FSM and NLG.	600	600	600	600
maxSimNumNLG	Maximum number of fitness evaluations for a given run for NLG.	305	306	303	303
maxSimNumFSM	Maximum number of fitness evaluations for a given run for FSM.	305	354	351	338
maxSimNumNTSS	Maximum number of fitness evaluations for for NTSS	150	150	150	150
maxIterNum	Maximum number of iterations allowed for an algorithmfor all Simulink models excluding FSM and NLG.	3500	3500	3500	3500
maxIterNumFSM	Maximum number of iterations allowed for an algorithm for FSM	800	800	800	800
maxIterNumNLG	Maximum number of iterations allowed for an algorithm for NLG	1200	1200	1200	1200
maxIterNumNTSS	Maximum number of iterations allowed for NTSS	1150	-	-	-
testSize	Percentage of the initialSimNum number of data used as a test data to evaluate the performance of surrogate models	20%	20%	20%	20%
trainSize	Number of simulation data in the dataset excluding test data used to train the surrogate models	80%	80%	80%	80%
samplingStrategy	Sampling strategy required by SMOTE to balance the dataset	minority	minority	minority	minority
k	Number of data points randomly generated to calculate their Euclidean distance from the logistic regression plane	–	–	5	–
ϵ	Percentage of margin around the constant c	–	–	–	5%

Table 3: Parameter names, descriptions and values used by SoTA algorithms for our 4 CI subjects.

Parameter Name	Description	Value
maxdepth	Maximum depth of the decision tree	5
minsampleleaf	Minimum number of samples in a leaf node	1
minsamplesplit	Minimum number of samples required to split an internal node	2
classweight	Weights associated to each class (fail and pass)	Inverse of the number of instances in each class

Table 4: Names, descriptions, the number of requirements of our case studies and the identifiers. For each case study, we indicate if the case study is computer-intensive (CI).

Name	Description	#Reqs	CI	ID
Tustin	A common flight control utility for computing the Tustin Integration – A Simulink model with 57 blocks.	9	✗	TU1…TU9
Regulator	A regulators inner loop architecture used in many feedback control applications – A Simulink model with 308 blocks.	1	✗	REG
Nonlinear Guidance	A nonlinear algorithm for generating a guidance command for an air vehicle – A Simulink model with 373 blocks.	1	✗	NLG
Finite State Machine	A finite state machine to enable autopilot mode if a hazardous situation is identified – A Simulink model with 303 blocks.	1	✗	FSM
Autopilot	A single-engine, high-wing, propeller-driven aircraft simulation with all six degrees of freedom – A Simulink model with 1549 blocks.	3	✓	AP1, AP2, AP3
Network Traffic Shaping System	An NTSS simulator [26] developed using three virtual machines and based on OpenWRT [?]	1	✓	NTSS

specifications, among which six could not be used for our approach because the requirements given for those six specifications either do not fail, or when they fail we cannot find any passing test input, i.e., they fail for all test inputs. For the latter cases, the failure models can be trivially defined as the entire input space. In our experiments, we consider form Lockheed's benchmark five Simulink specifications. These specifications have a total of 15 requirements combined for which we can generate both pass and fail test inputs.

In total, we have 16 requirements: one for NTSS, and 15 for the five Simulation specifications listed above. As discussed in Section 3, we define one fitness function per requirement. Hence, in total we have 16 different subjects for our evaluation. Among these, four are compute-intensive (CI) and twelve are non-CI. Both NTSS and autopilot are CI: On average, each execution of NTSS takes $\approx$ 4.5min, and each execution of autopilot takes $\approx$ 0.5min. The execution times for non-CI subjects are negligible (below one second). All the executions were on a machine with a 2.5 GHz Intel Core i9 CPU and 64 GB of DDR4 memory.

RQ1-Configuration. We compare eight versions of the surrogate-assisted algorithm. Seven are Algorithm 2 used with an individual surrogate model from those in Table 1. We refer to each of these algorithms as SA-XX where XX is the name of the surrogate model from Table 1. For example, SA-NN refers to Algorithm 2 used with NN. The final (i.e. eighth) algorithm is the dynamic surrogate-assisted one (Algorithm 3). We refer to Algorithm 3 as SA-DYN.

For RQ1, we use the 12 non-CI subjects in Table 4. Performing RQ1 experiments on CI subjects would be prohibitively expensive. For example, a ballpark estimate for the execution time of the experiments required to answer RQ1 is more than a year if the experiments are performed on NTSS using the same experimentation platform. Therefore, we select the non-CI subjects for RQ1.

Setting. For each subject, we run the eight SA-XX algorithms for an equal time budget. The time budget given for each subject depends on the subject execution time. The detailed time budgets for different surrogate-assisted algorithms are described in Table 5. To account for randomness, we repeat each algorithm for each subject ten times.

Table 5: Time budget (in minutes) allocated to surrogate-assisted algorithms (SA-XX) for non-CI subjects (12 subjects).

Simulink	Preprocessing	Main Loop	Total
TU1...TU9	27 min	28 min	55 min
REG	19 min	21 min	40 min
NLG	8 min	9 min	17 min
FSM	4 min	4 min	8 min

Metrics. To measure efficiency, we take the cardinality of the generated dataset, *DS*. Since the algorithms have the same time budget, an algorithm is more efficient than another if it generates a larger dataset. Recall that the output of the main loop in Figure 1 for surrogate-assisted algorithms is a set *DS* where the test inputs are either labelled by predicted or actual fitness values. To build failure models, we label test inputs as pass or fail based on these fitness values. To measure the accuracy of the pass and fail labels, we execute the predicted tests in *DS* to compute their actual fitness values. We then count the *incorrectly labeled* tests as a measure of

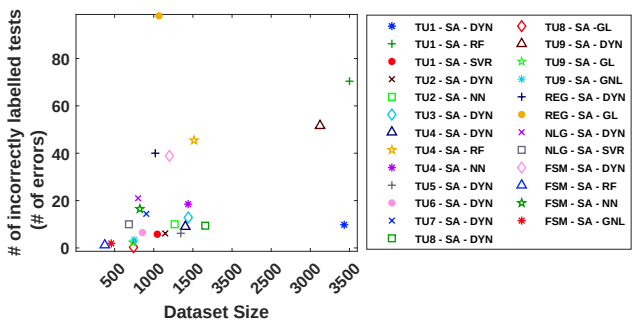


Figure 7: Comparing datasets generated by eight different surrogate-assisted algorithms with respect to the number of errors in the datasets and the dataset size.

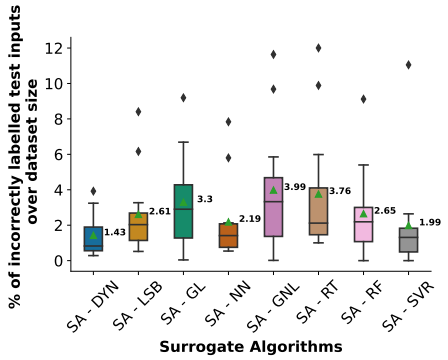


Figure 8: Percentages of the incorrectly labelled tests over the dataset size for different surrogate-assisted algorithms.

the accuracy of *DS*. That is, we count the number of tests in *DS* that have inconsistent pass-fail labels based on their predicted and actual fitness values.

Results. The scatter plot in Figure 7 shows the results of RQ1 experiments. The x-axis indicates $|DS|$ (i.e., our efficiency metric) and the y-axis indicates the number of incorrectly labelled tests in *DS* (i.e., our accuracy metric). Each point shows the result of applying one SA-XX algorithm to one subject. The 12 subjects are indicated by TU1 to TU9, REG, NLG, and FSM. For each subject, a better algorithm is the one that has high accuracy (i.e., low value on y-axis) and high efficiency (i.e., high value on x-axis). Since we compare eight algorithms for 12 subjects, we would need 96 points to show all the results. To reduce clutter, for each subject, we only show the Pareto Frontier (PF) points. That is, for each subject, we only show the algorithms that are not dominated by other algorithms either in terms of accuracy or in terms of efficiency. For example, for subject TU1, algorithms SA-DYN, SA-RF and SA-SVR dominate other algorithms and offer the best trade-offs between efficiency and accuracy. As Figure 7 shows, for four subjects, SA-DYN is the only best trade-off (i.e., PF point), and for eight subjects, it is one of the best PF points. For the latter eight cases, SA-DYN offers an alternative that, compared to the other PF points, is either considerably more accurate than others while its dataset is not much smaller; or its dataset is considerably larger while its accuracy is not much worse. For example, for TU1, SA-DYN, compared to SA-RF, provides better accuracy (60 less incorrect labels), while the dataset sizes are almost the same (3433 for SA-DYN vs. 3500 for SA-RF). Also, compared to SA-SVR, SA-DYN provides larger datasets (3433 vs. 1045) while the accuracy is almost the same (10 vs. 6 incorrect labels).

Table 6: Comparing dynamic surrogate-assisted (SA-DYN) with seven other surrogate-assisted algorithms with respect to Wilcoxon rank sum test [35] and the Vargha-Delaney's $\hat{A}_{12}$ effect size [46] for dataset size and percentage of incorrect labels over dataset size for all non-CI subjects (12 subjects).

Metric: Dataset Size			
Comparison	p value	A_{12} estimate	Effect size
SA-DYN vs SA-GL	3.70E-08	0.6012	Small
SA-DYN vs SA-GNL	1.58E-07	0.6007	Small
SA-DYN vs SA-LSB	1.58E-14	0.67	Medium
SA-DYN vs SA-NN	0.00558	0.5772	Small
SA-DYN vs SA-RT	3.86E-10	0.6410	Small
SA-DYN vs SA-RF	3.65E-07	0.6233	Small
SA-DYN vs SA-SVR	3.39E-16	0.7855	Large
Metric: Percentage of incorrect labels over dataset size			
Comparison	p value	A_{12} estimate	Effect size
SA-DYN vs SA-GL	2.14E-05	0.1802	Large
SA-DYN vs SA-GNL	1.27E-06	0.1648	Large
SA-DYN vs SA-LSB	0.000623	0.2533	Large
SA-DYN vs SA-NN	0.016336	0.4254	Small
SA-DYN vs SA-RT	2.21E-06	0.32375	Medium
SA-DYN vs SA-RF	0.026339	0.36535	Small
SA-DYN vs SA-SVR	0.01952	0.4961	Negligible

Figure 8 shows the distributions of the ratios of the number of incorrectly labelled tests over $|DS|$ for different SA algorithms and for all the twelve subjects. The SA-DYN algorithm has the lowest average ratio. This average ratio is 28% lower than that of the second best algorithm, SA-SVR, i.e., $\frac{1.99-1.43}{1.99} = 28\%$. We compared the results in Figure 8 using the non-parametric pairwise Wilcoxon rank sum test [35] with the level of significance (α) set to 0.01 and the Vargha-Delaney's $\hat{A}_{12}$ effect size [32] (See Table 6). The SA-DYN algorithm is statistically significantly better than other algorithms with a high effect size for GL, GNL and LSB; a medium effect size for RT, and a small effect size for NN and RF, and a negligible effect size for SVR. The comparison of the dataset sizes shows that SA-DYN generates datasets that are significantly larger than those generated by SVR with a large effect size. Further, SA-DYN generates significantly larger datasets that are at least 33% larger than those obtained from other algorithms (Figure 9). The statistical tests comparing the dataset size between SA-DYN with the individual SA algorithms is described in Table 6. Specifically, there is a significant difference between the dataset size generated by SA-DYN and other algorithms with a large effect size for SVR, medium effect size for LSB and small effect size for GPR-L, GPR-NL, NN, RT and RF.

The answer to RQ1 is that the dynamic surrogate-assisted algorithm is significantly more accurate and generates significantly larger datasets compared to using the surrogate-assisted algorithm with the seven surrogate models in Table 1 individually.

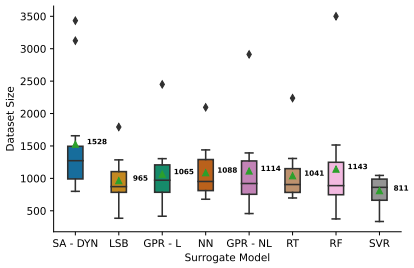


Figure 9: Dataset size comparison for dynamic surrogate-assisted algorithm and seven individual surrogate-assisted algorithms.

RQ2-Effectiveness. We compare SA-DYN, i.e., the best approach identified in RQ1, with the ML-guided techniques described in Section 3.2. In addition, we consider a standard random search algorithm as an additional baseline. In total, we compare SA-DYN with three algorithms for RQ2. In the remainder of this section, we refer to SA-DYN as SA. Further, we use RT and LR to refer to ML-guided techniques based on regression tree (Algorithm 4) and logistic regression (Algorithms 5), respectively, and RS for random search.

Setting. We apply the four algorithms to the 16 subjects in Table 4. For each CI subject, we execute the four algorithms for an equal time budget and then compare their results. But for the non-CI subjects, we should perform the comparison so that the results are meaningful for CI subjects since our work essentially targets CI systems. Note that while the criterion of fixing the time budget is reasonable for CI subjects, for the non-CI subjects, this criterion may favour RS since the three other algorithms have an additional overhead for training ML models. The overhead time is negligible compared to the subject execution time for CI subjects. But for non-CI subjects, we can execute many test inputs within the overhead time. This can impact the results in favour of RS. Therefore, in order to experiment with non-CI subjects and still be able to draw conclusions that are valid for CI subjects, we follow the approach proposed by Menghi et. al. [36], and use the execution time of CI subjects to limit the number of test inputs each algorithm executes for non-CI objects. Briefly, to compare algorithms with different overhead times, we allow the algorithm with the lower overhead time to execute x additional test inputs such that x multiplied by the execution time of a typical CI subject (instead of a non-CI subject) is equal to the difference in the algorithms' overhead time. Specifically, for the non-CI Simulink subjects in Table 4, we use the average execution time of the Simulink CI subject, i.e., autopilot. Given a time budget, we compute the maximum number of test executions that each of the SA, LR, RT, and RS algorithms can perform within this time budget for autopilot. We then use these numbers to cap the number of test executions for each algorithm when we compare them for the non-CI models in Table 4. The maximum number of test executions we used to compare them for non-CI subjects are also described in Table 8. Further, the time budget set to compare these algorithms for CI subjects are described in Table 7. Further, We repeated each algorithm for each subject except for NTSS twenty times, and ten times or NTSS due to its large execution time.

Table 7: Time taken for surrogate-assisted (SA), ML-guided test generation (LR and RT) and random baseline (RS) in terms of minutes for CI subjects (4 subjects).

Subject: AP1			
Algorithm	Preprocessing	Main Loop	Total
SA-DYN	71	73	144
RS	71	73	144
RT	71	73	144
LR	71	73	144

Subject: AP2			
Algorithm	Preprocessing	Main Loop	Total
SA-DYN	71	73	144
RS	71	73	144
RT	71	73	144
LR	71	73	144

Subject: AP3			
Algorithm	Preprocessing	Main Loop	Total
SA-DYN	121	134	255
RS	121	134	255
RT	121	134	255
LR	121	134	255

Subject: NTSS			
Algorithm	Preprocessing	Main Loop	Total
SA-DYN	450	250	700
RS	450	250	700
RT	450	250	700
LR	450	250	700

Metrics. We use the datasets obtained by SA, LR, RT, and RS to build decision rule models (DRM). To ensure that DRMs are not biased towards any of our four algorithms, we took the union of the datasets obtained from these four algorithms for tuning. We tuned the hyper-parameters of DRMs using the Bayesian optimization. Having fixed the parameter values, we trained a DRM separately for each dataset obtained from each repeat of our four algorithms. We evaluate the DRMs using three metrics: *accuracy*, *precision* and *recall*. Accuracy is the number of correctly predicted tests over the total number of tests. Since DRMs are mainly used to predict the failure class, we compute precision and recall for the failure class as follows: Precision is the number of fail class predictions that actually belong to the fail class, and recall is the number of fail class predictions made out of all the actual fail tests in the dataset. We use a test set to measure the accuracy, precision and recall of each DRM. Using random search, we generate a test set of size 600 for each Simulink subject and a test set of size 300 for NTSS.

Features for learning. For the Simulink subjects, we use as features the individual input variables of each subject model but exclude the following two kinds of variables: (1) Variables that are explicitly fixed to a value in a requirement (e.g., variable APEng discussed in Section 3.3). (2) Reference variables that indicate the desired value of a controlled process, noting that the system is expected to

Table 8: Time taken for surrogate-assisted (SA), ML-guided test generation (LR and RT) and random baseline (RS) in terms of number of simulations for non-CI subjects (12 subjects).

Subject: TU1...TU9			
Algorithm	Preprocessing	Main Loop	Total
SA-DYN	300	300	600
RS	300	319	619
RT	300	317	617
LR	300	308	608
Subject: REG			
Algorithm	Preprocessing	Main Loop	Total
SA-DYN	300	300	600
RS	300	348	648
RT	300	340	640
LR	300	329	629
Subject: NLG			
Algorithm	Preprocessing	Main Loop	Total
SA-DYN	300	5	305
RS	300	6	306
RT	300	3	303
LR	300	2	302
Subject: FSM			
Algorithm	Preprocessing	Main Loop	Total
SA-DYN	300	5	305
RS	300	54	354
RT	300	51	351
LR	300	38	338

satisfy its requirements for *any* value in a reference variable's valid range. As such, reference variables cannot contribute to bringing about failure conditions. The desired altitude variable discussed in Section 3.3 is an example of a reference variable.

For NTSS, as discussed in Section 2, we consider alternative features as follows: the set of all individual variables (i.e., individual NTSS classes), sums of two variables, sums of three variables, ..., and the sum of all eight variables. We then create, for each input feature, one DRM based on a dataset obtained by each of our four algorithms, i.e., SA, LR, RT and RS. In total, for each algorithm, we create 248 DRMs for NTSS. That is, the sets of all subsets larger than two (247) and the set of all individual variables. Given the large number of hypothesized input features, we evaluate the accuracy of the resulting DRMs and keep the input features that yield reasonably high accuracy over multiple runs of SA, LR, RT, RS. We set the accuracy threshold at 80%. In total, we retain two input features for NTSS and use them for comparing our four algorithms.

Results. Figures 15(a)-(c) compare the average accuracy of DRMs obtained by SA, LR and RT (on y-axis) with the average accuracy of DRMs obtained by RS (on x-axis) for our 16 subjects. Each point in each of Figures 15(a)-(c) corresponds to one study subject. Any point positioned above (resp. below) the dashed-dotted line indicates that the average accuracy of DRMs obtained by one of SA, LR or RT for one

Table 9: Comparing surrogate-assisted (SA), ML-guided (LR and RT) and random baseline (RS) algorithms with 50% execution time budget using Wilcoxon rank sum test [35] and the Vargha-Delaney's $\hat{A}_{12}$ effect size [46] for accuracy, recall and precision for all the subjects.

Metric: Accuracy			
Comparison	p value	A_{12} estimate	Effect size
SA vs RS	0.2920	0.5531896	Negligible
LR vs RS	0.2120	0.5577	Negligible
RT vs RS	0.1550	0.5567	Negligible
SA vs LR	0.5567	0.4962	Negligible
SA vs RT	0.6769	0.4964	Negligible
Metric: Recall			
Comparison	p value	A_{12} estimate	Effect size
SA vs RS	6.98E-06	6.98E-06	Small
LR vs RS	6.52E-09	0.3980	Small
RT vs RS	7.87E-05	0.4197	Small
SA vs LR	0.4196	0.51951	Negligible
SA vs RT	0.260	0.4975	Negligible
Metric: Precision			
Comparison	p value	A_{12} estimate	Effect size
SA vs RS	0.2722	0.4962	Negligible
LR vs RS	0.1655	0.5088	Negligible
RT vs RS	0.1695	0.4998	Negligible
SA vs LR	0.7691	0.4888	Negligible
SA vs RT	0.6565	0.4965	Negligible

of our subjects is higher (resp. lower) than the accuracy of DRMs obtained by RS. A blue point indicates that the difference in accuracy is statistically significant based on the Wilcoxon rank sum test. The DRMs obtained using SA are significantly more accurate than those obtained using RS for 14 subjects including all the CI subjects. The accuracy of the DRMs obtained using LR is significantly better than those obtained using RS for seven subjects. Finally, the accuracy of the DRMs obtained using RT is significantly better than those obtained using RS for nine subjects.

The average accuracy for different algorithms and different subjects is shown in Table 20. Overall for all the subjects, SA has the highest average accuracy (83%), followed by RT and LR with average accuracies of 78% and 76%, respectively. Finally, RS has the lowest average accuracy (68%). The pairwise statistical test result comparing different algorithms in terms of accuracy is described in Table 14. The accuracy for SA, RT and LR is significantly better than that of RS. The effect size for SA versus RS is medium, and the effect size for RT and LR versus RS is small.

Figure 21 compares the recall and precision for all the DRMs obtained using the datasets generated by SA, LR, RT and RS for our 16 subjects. A high recall indicates that the DRM characterizes the failure conditions with a high level of accuracy, and a high precision shows that the DRM generates failure instances with a high accuracy. The average recall and precision for different algorithms and different subjects is shown in Table 20. The SA

Table 10: Comparing surrogate-assisted (SA), ML-guided (LR and RT) and random baseline (RS) algorithms with 60% execution time budget using Wilcoxon rank sum test [35] and the Vargha-Delaney's $\hat{A}_{12}$ effect size [46] for accuracy, recall and precision for all the subjects.

Metric: Accuracy			
Comparison	p value	A_{12} estimate	Effect size
SA vs RS	9.74E-10	0.6168	Small
LR vs RS	0.9300	0.5372	Negligible
RT vs RS	0.0154	0.5608	Negligible
SA vs LR	6.73E-10	0.5716	Negligible
SA vs RT	1.56E-05	0.5522	Negligible
Metric: Recall			
Comparison	p value	A_{12} estimate	Effect size
SA vs RS	1.38E-06	0.4126	Medium
LR vs RS	7.11E-06	0.4287	Negligible
RT vs RS	1.66E-07	0.4100	Small
SA vs LR	0.4709	0.4872	Negligible
SA vs RT	0.8566	0.5011	Negligible
Metric: Precision			
Comparison	p value	A_{12} estimate	Effect size
SA vs RS	3.43E-15	0.5870	Small
LR vs RS	0.7619	0.4990	Negligible
RT vs RS	0.0023	0.5204	Negligible
SA vs LR	2.84E-13	0.5877	Small
SA vs RT	1.22E-09	0.5674	Negligible

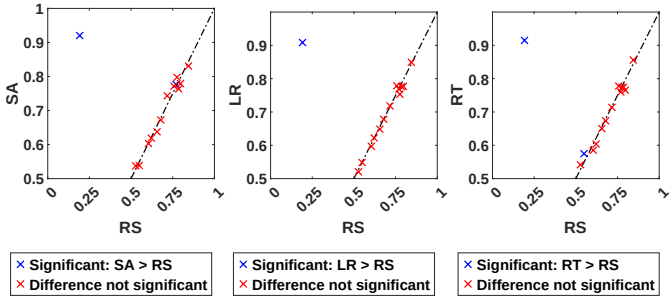


Figure 10: Comparing the accuracy of decision rule models obtained based on the datasets generated by SA, LR, and RT with those obtained by RS for the 16 subjects in Table 4 using 50% execution time budget.

algorithm has the highest average recall (88%), followed by LR (87%) and RT (85%), and RS has the lowest average recall (82%). Moreover, SA has the highest average precision (72%), followed by RT (64%) and LR (62%) and RS has the lowest average precision (57%). The pairwise statistical test result comparing different algorithms in terms of precision and recall is described in Table 14. Specifically, the recall for SA, RT and LR are significantly better than that of RS with a medium effect size for SA, a small effect size for LR, and negligible effect size for RT. Likewise, the precision for SA, RT and LR

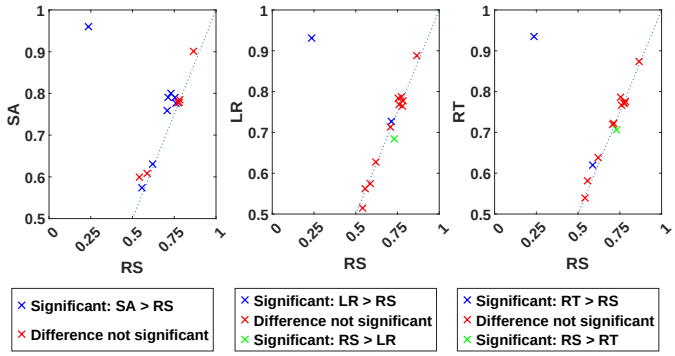


Figure 11: Comparing the accuracy of decision rule models obtained based on the datasets generated by SA, LR, and RT with those obtained by RS for the 16 subjects in Table 4 using 60% execution time budget.

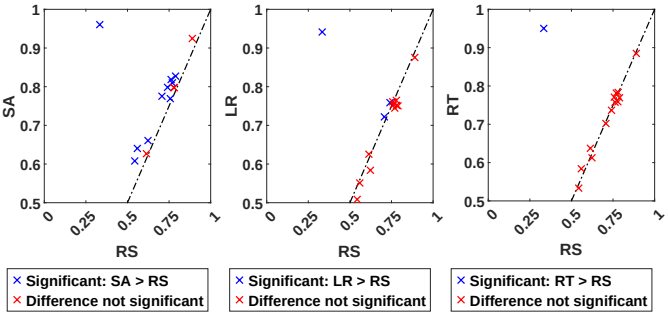


Figure 12: Comparing the accuracy of decision rule models obtained based on the datasets generated by SA, LR, and RT with those obtained by RS for the 16 subjects in Table 4 using 70% execution time budget.

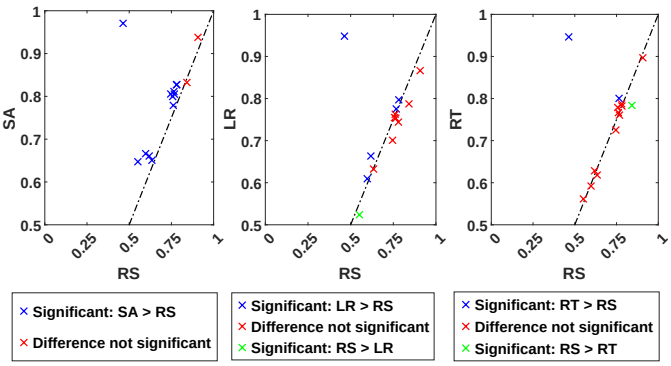


Figure 13: Comparing the accuracy of decision rule models obtained based on the datasets generated by SA, LR, and RT with those obtained by RS for the 16 subjects in Table 4 using 80% execution time budget.

Table 11: Comparing surrogate-assisted (SA), ML-guided (LR and RT) and random baseline (RS) algorithms with 70% execution time budget using Wilcoxon rank sum test [35] and the Vargha-Delaney's $\hat{A}_{12}$ effect size [46] for accuracy, recall and precision for all the subjects.

Metric: Accuracy			
Comparison	p value	A_{12} estimate	Effect size
SA vs RS	2.35E-23	0.6547	Small
LR vs RS	0.2942	0.5209	Negligible
RT vs RS	0.1293	0.5587	Negligible
SA vs LR	2.29E-23	0.6298	Small
SA vs RT	3.82E-18	0.5999	Small
Metric: Recall			
Comparison	p value	A_{12} estimate	Effect size
SA vs RS	1.83E-06	0.4137	Small
LR vs RS	0.0013	0.4523	Negligible
RT vs RS	1.52E-05	0.4276	Negligible
SA vs LR	0.9948	0.4699	Negligible
SA vs RT	0.9962	0.4907	Negligible
Metric: Precision			
Comparison	p value	A_{12} estimate	Effect size
SA vs RS	2.16E-23	0.6158	Small
LR vs RS	0.9713	0.4918	Negligible
RT vs RS	0.0899	0.5124	Negligible
SA vs LR	1.25E-21	0.6196	Small
SA vs RT	1.65E-18	0.6083	Small

Table 12: Comparing surrogate-assisted (SA), ML-guided (LR and RT) and random baseline (RS) algorithms with 80% execution time budget using Wilcoxon rank sum test [35] and the Vargha-Delaney's $\hat{A}_{12}$ effect size [46] for accuracy, recall and precision for all the subjects.

Metric: Accuracy			
Comparison	p value	A_{12} estimate	Effect size
SA vs RS	3.05E-26	0.6725	Medium
LR vs RS	0.0005	0.4755	Negligible
RT vs RS	0.0750	0.5549	Negligible
SA vs LR	1.30E-29	0.6695	Medium
SA vs RT	1.06E-20	0.6109	Small
Metric: Recall			
Comparison	p value	A_{12} estimate	Effect size
SA vs RS	0.0071	0.4475	Negligible
LR vs RS	0.7368	0.4990	Negligible
RT vs RS	8.82E-07	0.4003	Small
SA vs LR	0.1266	0.4591	Negligible
SA vs RT	0.0003	0.5508	Negligible
Metric: Precision			
Comparison	p value	A_{12} estimate	Effect size
SA vs RS	1.37E-24	0.6389	Small
LR vs RS	0.0048	0.4694	Negligible
RT vs RS	0.0209	0.5283	Negligible
SA vs LR	1.31E-27	0.6678	Medium
SA vs RT	4.43E-17	0.6141	Small

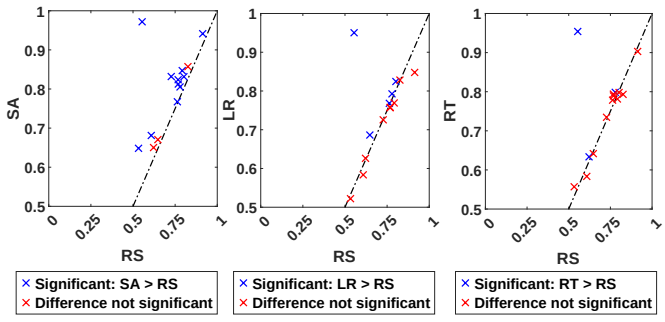


Figure 14: Comparing the accuracy of decision rule models obtained based on the datasets generated by SA, LR, and RT with those obtained by RS for the 16 subjects in Table 4 using 90% execution time budget.

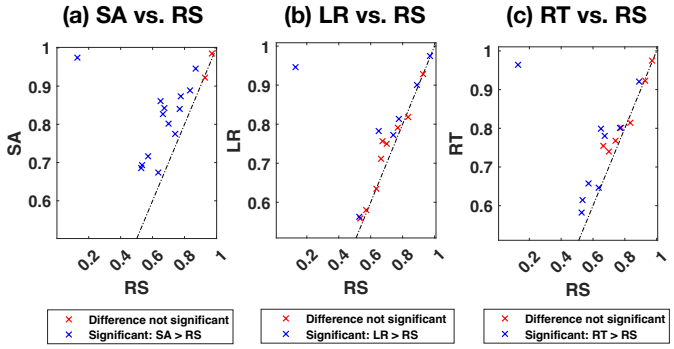


Figure 15: Comparing the accuracy of decision rule models obtained based on the datasets generated by SA, LR, and RT with those obtained by RS for the 16 subjects in Table 4 using 100% execution time budget.

are significantly better than that of RS with a medium effect size for SA, small effect size for RT and negligible effect size for LR. Finally, the accuracy, recall and precision values for SA are significantly higher than those for RT and LR.

The answer to RQ2 is that the DRMs obtained by SA for 16 subjects have accuracy, precision and recall values that are significantly higher than those of the DRMs obtained by the other three algorithms (i.e., RT, LR and RS).

Table 13: Comparing surrogate-assisted (SA), ML-guided (LR and RT) and random baseline (RS) algorithms with 90% execution time budget using Wilcoxon rank sum test [35] and the Vargha-Delaney's $\hat{A}_{12}$ effect size [46] for accuracy, recall and precision for all the subjects.

Metric: Accuracy			
Comparison	p value	A_{12} estimate	Effect size
SA vs RS	1.37E-24	0.6389	Small
LR vs RS	0.0626	0.4833	Negligible
RT vs RS	0.2019	0.5448	Negligible
SA vs LR	1.54E-31	0.6630	Small
SA vs RT	1.38E-22	0.6210	Small
Metric: Recall			
Comparison	p value	A_{12} estimate	Effect size
SA vs RS	0.0002	0.4385	Negligible
LR vs RS	0.0057	0.4654	Negligible
RT vs RS	3.76E-05	0.4332	Negligible
SA vs LR	0.6330	0.4759	Negligible
SA vs RT	0.5489	0.5052	Negligible
Metric: Precision			
Comparison	p value	A_{12} estimate	Effect size
SA vs RS	2.80E-29	0.6600	Small
LR vs RS	0.4744	0.4888	Negligible
RT vs RS	0.1339	0.5287	Negligible
SA vs LR	1.51E-28	0.6775	Medium
SA vs RT	4.77E-21	0.6377	Small

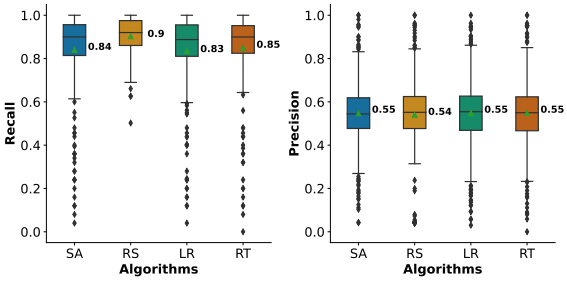


Figure 16: Recall and Precision for the DRMs obtained from the datasets generated by SA, RL, RT and RS for all 16 subjects using 50% execution time budget.

Table 14: Comparing surrogate-assisted (SA), ML-guided (LR and RT) and random baseline (RS) algorithms with 100% execution time budget using Wilcoxon rank sum test [35] and the Vargha-Delaney's $\hat{A}_{12}$ effect size [46] for accuracy, recall and precision for all the subjects.

Metric: Accuracy			
Comparison	p value	A_{12} estimate	Effect size
SA vs RS	3.04E-43	0.7264	Medium
LR vs RS	0.00013	0.59383	Small
RT vs RS	2.01E-18	0.6383	Small
SA vs LR	1.16E-40	0.6640	Small
SA vs RT	1.45E-31	0.6171	Small
Metric: Recall			
Comparison	p value	A_{12} estimate	Effect size
SA vs RS	1.89E-12	0.7319	Medium
LR vs RS	4.64E-11	0.6323	Small
RT vs RS	0.003872	0.5716	Negligible
SA vs LR	0.679912	0.4863	Negligible
SA vs RT	5.78E-05	0.5606	Negligible
Metric: Precision			
Comparison	p value	A_{12} estimate	Effect size
SA vs RS	8.70E-42	0.7108	Medium
LR vs RS	2.11E-05	0.5722	Negligible
RT vs RS	1.29E-15	0.6073	Small
SA vs LR	1.57E-39	0.7092	Medium
SA vs RT	7.91E-32	0.6603	Small

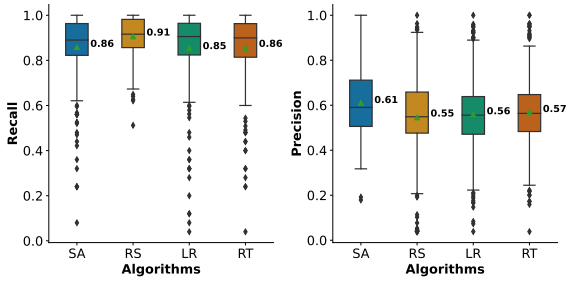


Figure 17: Recall and Precision for the DRMs obtained from the datasets generated by SA, RL, RT and RS for all 16 subjects using 60% execution time budget.

RQ3-SOTA Comparison We compare SA – the top-performing algorithm from RQ2 – with an adaptation of Alhazen to numeric-input systems. We hereafter use SoTA to refer to this adaptation, discussed next. Similar to SA, in SoTA, we generate test inputs according to the description in Section 4 (Study Subjects). The workflow of SoTA then matches the steps in Figure 1 with the difference that the model trained and refined in the main loop (i.e. step 3) is always a decision tree model; this decision tree is returned as the failure model at the end. Recall that our approach has a separate step, i.e., step 5, after the main loop to build failure models from the data generated by different algorithms. This step is not required in SoTA, noting that the model that SoTA refines during its main loop is used as the failure model. Algorithm 6 shows our implementation of SoTA. SoTA starts from an initial dataset (line 8). In each iteration, it builds a decision tree on the dataset (line 10). It then generates test inputs using all the paths in the decision tree (lines 11-17). These test inputs are executed and added to the dataset along with their labels (line 18). The final decision tree is returned on line 21.

Setting. For this comparison, we apply SoTA to our CI-subjects in Table 4, i.e., NTSS, AP1, AP2 and AP3. For the decision tree parameters, e.g. maximum depth of tree and class weight, we use the same parameters as in the original study [27]. We execute SoTA

Table 15: Average accuracy, recall and precision over all the runs of surrogate-assisted (SA), ML-guided (LR and RT) and random baseline (RS) for all 16 subjects with 50% execution time budget.

Metric: Accuracy				
Subject	SA	RS	LR	RT
TU1	0.92	0.19	0.91	0.91
TU2	0.78	0.77	0.77	0.76
TU3	0.78	0.80	0.78	0.77
TU4	0.76	0.79	0.78	0.77
TU5	0.74	0.72	0.72	0.71
TU6	0.54	0.55	0.55	0.57
TU7	0.54	0.53	0.52	0.54
TU8	0.67	0.68	0.68	0.67
TU9	0.64	0.66	0.65	0.65
REG	0.60	0.60	0.60	0.59
AP1	0.83	0.84	0.85	0.86
AP2	0.77	0.76	0.78	0.78
AP3	0.62	0.62	0.62	0.60
NTSS	0.80	0.78	0.75	0.77
Average	0.71	0.66	0.71	0.71
Metric: Recall				
Subject	SA	RS	LR	RT
TU1	0.31	0.92	0.33	0.35
TU2	0.91	0.89	0.86	0.90
TU3	0.88	0.86	0.88	0.87
TU4	0.83	0.86	0.85	0.84
TU5	0.88	0.88	0.87	0.90
TU6	0.96	0.95	0.92	0.92
TU7	0.93	0.93	0.92	0.93
TU8	0.99	0.99	0.98	0.99
TU9	0.97	0.99	0.99	0.98
REG	0.89	0.88	0.89	0.91
AP3	0.80	0.81	0.77	0.82
AP1	0.90	0.91	0.88	0.90
AP2	0.71	0.89	0.78	0.78
NTSS	0.79	0.88	0.77	0.81
Average	0.84	0.90	0.83	0.85
Metric: Precision				
Subject	SA	RS	LR	RT
TU1	0.20	0.05	0.18	0.22
TU2	0.59	0.64	0.64	0.63
TU3	0.50	0.53	0.51	0.49
TU4	0.50	0.53	0.52	0.51
TU5	0.55	0.53	0.53	0.52
TU6	0.50	0.51	0.51	0.53
TU7	0.48	0.47	0.47	0.48
TU8	0.63	0.63	0.63	0.63
TU9	0.60	0.61	0.61	0.61
REG	0.56	0.56	0.56	0.54
AP3	0.48	0.48	0.49	0.47
AP1	0.85	0.85	0.88	0.88
AP2	0.62	0.57	0.62	0.61
NTSS	0.58	0.60	0.53	0.55
Average	0.55	0.54	0.55	0.55

Table 16: Average accuracy, recall and precision over all the runs of surrogate-assisted (SA), ML-guided (LR and RT) and random baseline (RS) for all 16 subjects with 60% execution time budget.

Metric: Accuracy				
Subject	SA	RS	LR	RT
TU1	0.96	0.24	0.93	0.93
TU2	0.78	0.77	0.79	0.77
TU3	0.78	0.78	0.76	0.77
TU4	0.78	0.78	0.78	0.78
TU6	0.57	0.56	0.56	0.58
TU7	0.60	0.54	0.51	0.54
TU5	0.76	0.71	0.71	0.72
TU8	0.80	0.73	0.68	0.71
TU9	0.79	0.71	0.73	0.72
REG	0.61	0.59	0.57	0.62
AP1	0.90	0.87	0.89	0.87
AP2	0.78	0.76	0.77	0.77
AP3	0.63	0.62	0.63	0.64
NTSS	0.79	0.75	0.78	0.79
Average	0.75	0.67	0.72	0.73
Metric: Recall				
Subject	SA	RS	LR	RT
TU1	0.49	0.88	0.40	0.46
TU2	0.87	0.89	0.90	0.89
TU3	0.87	0.90	0.88	0.88
TU4	0.86	0.87	0.86	0.85
TU6	0.95	0.94	0.94	0.92
TU7	0.90	0.92	0.94	0.93
TU5	0.87	0.90	0.91	0.88
TU8	0.99	0.99	0.97	0.98
TU9	0.99	0.99	0.97	0.99
REG	0.87	0.91	0.92	0.87
AP1	0.94	0.91	0.90	0.91
AP2	0.77	0.91	0.72	0.77
AP3	0.81	0.80	0.81	0.83
NTSS	0.81	0.86	0.81	0.83
Average	0.86	0.91	0.85	0.86
Metric: Precision				
Subject	SA	RS	LR	RT
TU1	0.56	0.05	0.30	0.34
TU2	0.65	0.64	0.62	0.65
TU3	0.51	0.50	0.50	0.49
TU4	0.53	0.53	0.51	0.52
TU6	0.52	0.51	0.50	0.53
TU7	0.52	0.48	0.47	0.48
TU5	0.57	0.52	0.53	0.54
TU8	0.74	0.67	0.64	0.66
TU9	0.72	0.65	0.64	0.65
REG	0.57	0.55	0.54	0.57
AP1	0.91	0.88	0.88	0.89
AP2	0.62	0.57	0.62	0.60
AP3	0.49	0.48	0.49	0.47
NTSS	0.60	0.55	0.56	0.57
Average	0.61	0.54	0.56	0.57

Table 17: Average accuracy, recall and precision over all the runs of surrogate-assisted (SA), ML-guided (LR and RT) and random baseline (RS) for all 16 subjects with 70% execution time budget.

Metric: Accuracy				
Subject	SA	RS	LR	RT
TU1	0.96	0.33	0.94	0.95
TU2	0.80	0.78	0.76	0.76
TU3	0.83	0.79	0.75	0.77
TU4	0.80	0.78	0.75	0.78
TU5	0.78	0.71	0.72	0.70
TU6	0.64	0.56	0.55	0.58
TU7	0.61	0.54	0.51	0.53
TU8	0.82	0.76	0.75	0.76
TU9	0.80	0.74	0.76	0.74
REG	0.66	0.62	0.58	0.61
AP1	0.92	0.89	0.88	0.88
AP2	0.77	0.76	0.76	0.77
AP3	0.63	0.61	0.63	0.64
NTSS	0.81	0.77	0.74	0.78
Average	0.77	0.69	0.72	0.73
Metric: Recall				
Subject	SA	RS	LR	RT
TU1	0.70	0.84	0.53	0.56
TU2	0.86	0.89	0.89	0.90
TU3	0.86	0.89	0.91	0.90
TU4	0.87	0.88	0.87	0.85
TU6	0.93	0.94	0.93	0.93
TU7	0.93	0.93	0.95	0.95
TU5	0.87	0.93	0.93	0.90
TU8	0.99	0.99	0.98	0.99
TU9	0.99	0.99	0.98	0.99
REG	0.82	0.88	0.91	0.89
AP1	0.94	0.92	0.92	0.90
AP2	0.82	0.90	0.75	0.81
AP3	0.83	0.82	0.82	0.80
NTSS	0.82	0.92	0.85	0.83
Average	0.87	0.91	0.87	0.87
Metric: Precision				
Subject	SA	RS	LR	RT
TU1	0.57	0.06	0.42	0.43
TU2	0.69	0.65	0.63	0.63
TU3	0.57	0.52	0.47	0.49
TU4	0.55	0.52	0.49	0.53
TU6	0.57	0.51	0.51	0.53
TU7	0.53	0.48	0.47	0.48
TU5	0.59	0.51	0.50	0.51
TU8	0.75	0.70	0.70	0.70
TU9	0.73	0.68	0.66	0.68
REG	0.62	0.58	0.55	0.57
AP1	0.94	0.90	0.89	0.92
AP2	0.60	0.57	0.60	0.60
AP3	0.49	0.48	0.49	0.49
NTSS	0.63	0.59	0.57	0.56
Average	0.63	0.55	0.57	0.58

Table 18: Average accuracy, recall and precision over all the runs of surrogate-assisted (SA), ML-guided (LR and RT) and random baseline (RS) for all 16 subjects with 80% execution time budget.

Metric: Accuracy				
Subject	SA	RS	LR	RT
TU1	0.97	0.46	0.95	0.95
TU2	0.80	0.76	0.75	0.78
TU3	0.83	0.78	0.74	0.79
TU4	0.81	0.77	0.75	0.80
TU6	0.67	0.60	0.61	0.59
TU7	0.65	0.55	0.52	0.56
TU5	0.81	0.75	0.70	0.73
TU8	0.83	0.78	0.80	0.78
TU9	0.81	0.77	0.78	0.76
REG	0.66	0.62	0.66	0.63
AP1	0.94	0.91	0.87	0.90
AP2	0.78	0.76	0.76	0.77
AP3	0.65	0.63	0.63	0.62
NTSS	0.83	0.84	0.79	0.78
Average	0.79	0.71	0.74	0.74
Metric: Recall				
Subject	SA	RS	LR	RT
TU1	0.75	0.83	0.68	0.63
TU2	0.87	0.81	0.90	0.89
TU3	0.87	0.80	0.90	0.90
TU4	0.87	0.81	0.89	0.87
TU6	0.93	0.81	0.94	0.93
TU7	0.91	0.83	0.96	0.91
TU5	0.86	0.81	0.90	0.89
TU8	0.99	0.99	0.98	0.98
TU9	1.00	0.99	0.97	0.99
REG	0.78	0.85	0.90	0.84
AP1	0.97	0.94	0.95	0.94
AP2	0.83	0.92	0.76	0.84
AP3	0.86	0.84	0.83	0.86
NTSS	0.86	0.87	0.79	0.81
Average	0.88	0.86	0.88	0.88
Metric: Precision				
Subject	SA	RS	LR	RT
TU1	0.65	0.08	0.46	0.41
TU2	0.68	0.63	0.62	0.65
TU3	0.57	0.50	0.47	0.51
TU4	0.57	0.50	0.49	0.55
TU6	0.59	0.54	0.50	0.54
TU7	0.56	0.49	0.47	0.50
TU5	0.64	0.56	0.51	0.53
TU8	0.77	0.72	0.69	0.72
TU9	0.74	0.70	0.68	0.70
REG	0.62	0.58	0.53	0.59
AP1	0.93	0.92	0.87	0.91
AP2	0.60	0.57	0.59	0.60
AP3	0.51	0.50	0.50	0.48
NTSS	0.62	0.63	0.58	0.56
Average	0.65	0.56	0.57	0.59

Table 19: Average accuracy, recall and precision over all the runs of surrogate-assisted (SA), ML-guided (LR and RT) and random baseline (RS) for all 16 subjects with 90% execution time budget.

Metric: Accuracy				
Subject	SA	RS	LR	RT
TU1	0.97	0.55	0.95	0.95
TU2	0.81	0.77	0.76	0.79
TU3	0.85	0.79	0.77	0.78
TU4	0.82	0.77	0.76	0.79
TU5	0.83	0.73	0.73	0.73
TU6	0.68	0.61	0.58	0.58
TU7	0.65	0.53	0.52	0.56
TU8	0.83	0.80	0.82	0.80
TU9	0.81	0.78	0.79	0.80
REG	0.67	0.65	0.69	0.64
AP1	0.94	0.91	0.85	0.90
AP2	0.77	0.76	0.77	0.78
AP3	0.65	0.62	0.63	0.63
NTSS	0.86	0.83	0.83	0.79
Average	0.80	0.72	0.75	0.75
Metric: Recall				
Subject	SA	RS	LR	RT
TU1	0.75	0.83	0.68	0.63
TU2	0.87	0.91	0.90	0.89
TU3	0.87	0.90	0.90	0.90
TU4	0.87	0.91	0.89	0.87
TU5	0.86	0.91	0.90	0.89
TU6	0.93	0.91	0.94	0.93
TU7	0.91	0.93	0.96	0.91
TU8	0.99	0.99	0.98	0.98
TU9	1.00	0.99	0.97	0.99
REG	0.78	0.85	0.90	0.84
AP1	0.97	0.94	0.95	0.94
AP2	0.83	0.92	0.76	0.84
AP3	0.86	0.84	0.83	0.86
NTSS	0.86	0.87	0.79	0.81
Average	0.88	0.91	0.88	0.88
Metric: Precision				
Subject	SA	RS	LR	RT
TU1	0.69	0.11	0.48	0.52
TU2	0.70	0.64	0.63	0.66
TU3	0.61	0.52	0.50	0.50
TU4	0.59	0.50	0.50	0.54
TU5	0.68	0.54	0.54	0.55
TU6	0.60	0.55	0.53	0.53
TU7	0.56	0.48	0.47	0.49
TU8	0.77	0.74	0.74	0.74
TU9	0.74	0.67	0.68	0.69
REG	0.65	0.60	0.55	0.60
AP1	0.94	0.92	0.85	0.91
AP2	0.59	0.57	0.60	0.60
AP3	0.51	0.48	0.49	0.48
NTSS	0.67	0.61	0.62	0.56
Average	0.66	0.57	0.58	0.60

Table 20: Average accuracy, recall and precision over all the runs of surrogate-assisted (SA), ML-guided (LR and RT) and random baseline (RS) for all 16 subjects with 100% execution time budget.

Metric: Accuracy				
Subject	SA	RS	LR	RT
TU1	0.97	0.66	0.95	0.96
TU2	0.84	0.77	0.79	0.80
TU3	0.84	0.67	0.76	0.78
TU4	0.86	0.65	0.78	0.80
TU5	0.83	0.66	0.71	0.75
TU6	0.69	0.53	0.56	0.61
TU7	0.69	0.53	0.56	0.58
TU8	0.99	0.97	0.97	0.98
TU9	0.92	0.93	0.91	0.92
REG	0.72	0.57	0.58	0.66
AP1	0.97	0.89	0.90	0.92
AP2	0.80	0.70	0.75	0.74
AP3	0.67	0.64	0.64	0.65
NLG	0.77	0.74	0.77	0.77
FSM	0.87	0.78	0.81	0.80
NTSS	0.89	0.83	0.82	0.81
Average	**0.83**	**0.72**	**0.77**	**0.78**
Metric: Recall				
Subject	SA	RS	LR	RT
TU1	0.88	0.87	0.70	0.64
TU2	0.84	0.83	0.92	0.88
TU3	0.89	0.86	0.90	0.88
TU4	0.86	0.82	0.88	0.86
TU5	0.88	0.85	0.91	0.89
TU6	0.95	0.84	0.96	0.91
TU7	0.91	0.75	0.94	0.91
TU8	0.99	0.98	0.98	0.97
TU9	0.99	0.98	0.98	0.98
REG	0.73	0.78	0.92	0.82
AP1	0.98	0.89	0.96	0.96
AP2	0.72	0.59	0.61	0.61
AP3	0.84	0.69	0.81	0.79
NLG	0.93	0.90	0.95	0.92
FSM	0.87	0.67	0.68	0.68
NTSS	0.86	0.90	0.86	0.82
Average	**0.88**	**0.82**	**0.87**	**0.85**
Metric: Precision				
Subject	SA	RS	LR	RT
TU1	0.71	0.04	0.44	0.57
TU2	0.76	0.65	0.60	0.68
TU3	0.60	0.43	0.50	0.51
TU4	0.67	0.42	0.55	0.56
TU5	0.67	0.50	0.53	0.57
TU6	0.61	0.50	0.51	0.55
TU7	0.59	0.47	0.49	0.51
TU8	0.91	0.91	0.85	0.91
TU9	0.90	0.91	0.84	0.90
REG	0.72	0.55	0.55	0.61
AP1	0.97	0.93	0.90	0.92
AP2	0.69	0.54	0.61	0.58
AP3	0.53	0.41	0.52	0.47
NLG	0.82	0.80	0.80	0.81
FSM	0.70	0.42	0.55	0.48
NTSS	0.70	0.62	0.63	0.60
Average	**0.72**	**0.57**	**0.62**	**0.64**

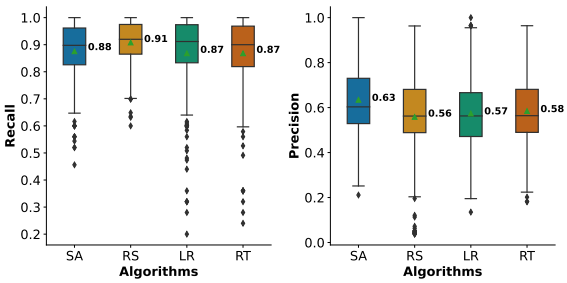


Figure 18: Recall and Precision for the DRMs obtained from the datasets generated by SA, RL, RT and RS for all 16 subjects using 70% execution time budget.

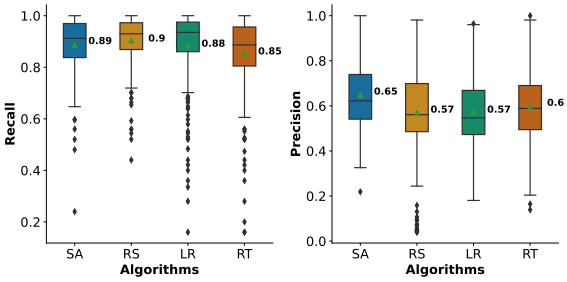


Figure 19: Recall and Precision for the DRMs obtained from the datasets generated by SA, RL, RT and RS for all 16 subjects using 80% execution time budget.

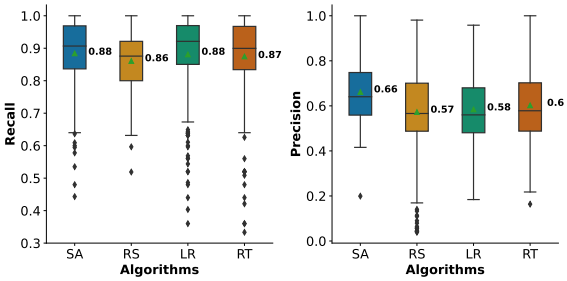


Figure 20: Recall and Precision for the DRMs obtained from the datasets generated by SA, RL, RT and RS for all 16 subjects using 90% execution time budget.

for the same time budget as SA. For SA, we use the dataset generated in RQ2. We repeat SoTA for twenty times for each subject except for NTSS. For NTSS, we repeat it only ten times due to the expensive execution time.

Metrics. In order to compare SoTA and SA, we build decision trees based on the datasets generated by SA in RQ2. To do so, we use the same decision tree parameters as those used by SoTA. We compare the decision trees using the three metrics explained in RQ2, i.e. accuracy, precision for fail class and recall for fail class. We also use the same test set utilized in RQ2.

Results. Figure 23 compares the average accuracy of the decision trees obtained by SA (on y-axis) against those obtained by SoTA (on x-axis) across the four CI subjects in Table 4. Similar to Figure 15,

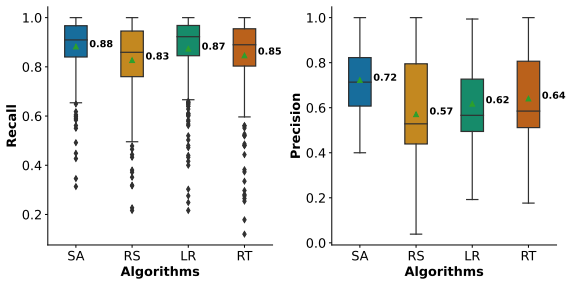


Figure 21: Recall and Precision for the DRMs obtained from the datasets generated by SA, RL, RT and RS for all 16 subjects using 100% execution time budget.

each point on Figure 23 corresponds to one study subject and a blue point denotes a statistically significant difference in accuracy, determined using the Wilcoxon rank-sum test. As figure 23 shows, the decision trees obtained using SA are significantly more accurate than those obtained by SoTA, for three out of the four subjects with a medium effect size. For the forth subject there is no statistically significant difference.

Figure 24 compares the average accuracy of the decision trees obtained using SA and those obtained by SoTA for the four CI subjects as the execution-time budget is varied from 50% to 100%. As the figure shows, the average accuracy of SA is consistently higher than SoTA as the time budget increases.

Finally, Figure 25 compares the recall and precision for all the decision trees obtained using SA and those obtained by SoTA for the four subjects. As shown by the figure, the average recall of SA (85%) is higher than SoTA (83%). Further, the recall of SA is significantly better than SoTA with small effect size. Moreover, the average precision of SA (76%) is higher than SoTA (73%). Similar to recall, the precision of SA is significantly better than SoTA with small effect size.

Our evaluation for RQ3 performed using dynamic surrogate-assisted and the state-of-the-art approach over CI subjects indicates that our dynamic surrogate-assisted approach yields failure models with higher accuracy, precision and recall compared to those obtained from the state-of-the-art approach.

RQ4-Usefulness. In view of the results of RQ2, we use the DRMs generated by the SA algorithm to evaluate usefulness. We focus on the DRMs for the CI subjects in Table 4, i.e., NTSS, AP1, AP2 and AP3. For these subjects, we can validate whether the inferred rules lead to genuinely spurious failures. For NTSS, we had direct access to domain experts, and for autopilot, we had detailed requirements and design documents. Recall that the rules we obtain from DRMs are in the form of IF-condition-THEN-prediction. Each rule has a confidence that shows what percentage of the tests satisfying the condition of the rule conform to the rule's prediction label. From the DRMs for each of the four CI subjects, we extract the rules that predict the fail class with a confidence of 100%. These rules are candidates for specifying spurious failures, since they identify conditions that lead to and only to failures. We then select the rules that are not subsumed by others through logical implication. We use the Z3 SMT solver [15] to find logical implications. In the end, we obtain seven rules for NTSS, 17 rules for AP1, 24 rules for AP2

```
1  Algorithm 6: SoTA
2  Input S: System
3  Input R = {R_1, ..., R_n}: Ranges for input variables v_1 to v_n
4  Input F: Fitness Function
5  Input Budget: Maximum number of fitness evaluations
6  Output DecisionTree: A decision tree model (failure model)
7
8  DS ← InitializeAndBalance(S, R, F, Budget/2);
9  while (|DS| ≤ Budget/2) && not (time budget expired) do
10 |  DecisionTree ← Train(DS);
11 |  Let P_1, ..., P_q be all the paths obtained from DecisionTree; //
   |      Extract all the paths from the decision tree
12 |  for each path P_k s.t. k ∈ {1, ..., q} do // Generate a test in
   |      each path based on the ranges obtained from that path
13 |  |  Let R'_{i1}, ..., R'_{im} be reduced ranges obtained from P_k;
14 |  |  for each variable v_{i_j} s.t. j ∈ {1, ..., m} do
15 |  |  |  R ← (R \ {R_{i_j}}) ∪ {R'_{i_j}}; // Replace the range R_{i_j} of v_{i_j} in
   |  |  |      R with the new reduced range R'_{i_j} from line 14
16 |  |  end
17 |  |  {t} ← GenerateTests(R, 1); //Generate a test in path P_k
18 |  |  DS ← DS ∪ Execute(S, {t}, F); // Compute fitness for t
   |  |      and add to DS
19 |  end
20 end
21 return DecisionTree
```

Figure 22: Our implementation of SoTA

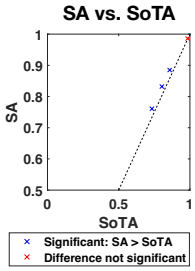
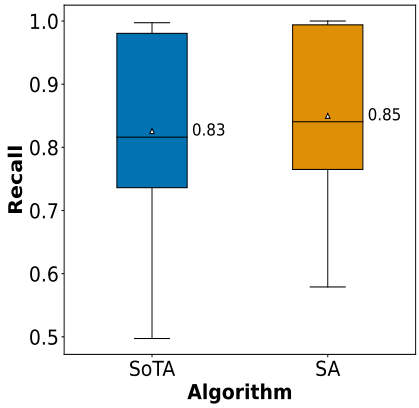


Figure 23: Comparing the accuracies of the decision trees obtained on the datasets generated by SA and the decision trees returned by SoTA for all the four CI subjects in Table 4.

Figure 24: Average accuracy of the decision trees obtained using SA and SoTA for the four CI subjects as the time budget is varied.

and 15 rules for AP3. On average, the rules for NTSS include two variables and three predicates, and the rules for autopilot include three variables and four predicates. Tables 21, 22, 23, 24 show the list of rules for NTSS, AP1, AP2, AP3, respectively.



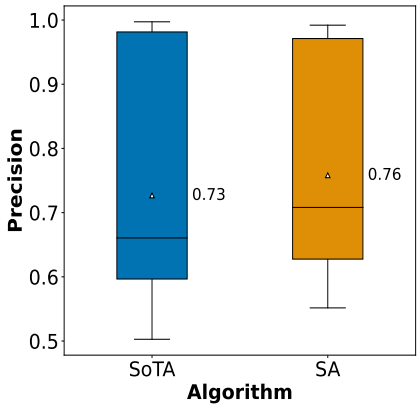


Figure 25: Recall and Precision for the decision trees obtained based on SA and SoTA for four subjects in Table 4.

Validating Obtained Rules. To validate the rules for NTSS, we present the rules to domain experts. Our domain expert for NTSS is a seasoned network technologist and software engineer with more than 25 years of experience. The expert has been using the core enabling technology of NTSS (CAKE [23], discussed earlier in this section) in commercial networking solutions since 2018. Among the seven rules for NTSS, one rule constrains an input feature formulating a sum of the NTSS input variables (This rule is in bold in Table 21. The rest of the six rules involve predicates over individual input variables. The domain expert approved all the rules as they all correspond to situations where NTSS is overloaded with large traffic volumes, and hence, poor network quality is expected. We note that, for each input variable of NTSS, there is a threshold that is determined by the NTSS setup configuration. Among the seven rules, six constrain input variables near these known threshold values. An example of such rules is: class6 $\geq$ 91%· thresh6 $\wedge$ class7 $\geq$ 87%· thresh7 THEN FAIL where thresh6 and thresh7 are thresholds of class6 and class7, respectively. These rules, while useful, could be estimated based on the

NTSS class thresholds, even though identifying the limits for individual variables (e.g., 91%· thresh6 or 87%· thresh7) still poses a challenge for domain experts. The expert indicated that the latter group of rules, while conforming to the expert's intuition, could not be predicted based on the existing thresholds or by their expertise-based testing. The expert justified the latter group of rules not based on threshold settings for individual NTSS classes but rather based on the internal system architecture and the priority relationships between different NTSS classes. The single rule that involves multiple input variables and is illustrated in Section 2 was of particular interest. This prompted an investigation into the source code of CAKE that confirmed the higher priority of NTSS class 7, 6 and 5 compared to other classes of NTSS, and hence, the need to cap the combined cumulative usage of these three classes, but not other combinations of NTSS classes.

For the three requirements of autopilot, we check the rules against the requirements and design documentation [7]. Specifically, AP1 states that *steady state roll commands shall be tracked within 1 degree*. The identified rules for this requirement which are shown in Table 22 mainly have TurnKnob. Based on our investigation of the autopilot handbook, TurnKnob controls the the roll of the airplane. APEng is also related to the engagement of autopilot. That is, if autopilot is not engaged (APEng = 0), the pilot is responsible for controlling the airplane. Therefore, if the pilot does not provide TurnKnob with appropriate values, then the requirement is failed. For example, in the first rule of Table 22, if the value of TurnKnob is greater than 4 and autopilot is not engaged (APEng = 0) then the requirement is failed. Moreover, ALTMode is related to maintaining the altitude of the airplane. In other words, when ALTMode is engaged (i.e. it is equal to 1), the aircraft maintains its altitude and therefore its altitude will not change. As can be seen in Table 22, rules 2 to 11, have APEng, ALTMode and TurnKnob. Although in only two out of 10 rules the value of ALTMode is 1, we believe this input variable is not directly related to the roll of the airplane. Finally, rules 15 and 16 in Table 22, have both TurnKnob and PitchWheel. These rules identify spurious failures related to when the position of the nose of the airplane (PitchWheel) and the roll of the airplane (TurnKnob). As can be seen in rules 15 and 16, the value of TurnKnob is relatively high and the value of PitchWheel varies from -11.63 to 5.96. This indicates that regardless of the position of the nose of the airplane (downward or upward), the requirement is failed when the airplane is turning sharply. Therefore, the main contributor to the failure of this requirement is the inappropriate value of TurnKnob. However, we are not able to confirm three rules that have PitchWheel, APEng and ALTMode (these rules are in blue and bold in Table 22) since based on our investigation, PitchWheel and ALTMode are related to the altitude of the airplane rather than its roll. Further expertise is needed to confirm whether the constraints over the predicates of PitchWheel, APEng and ALTMode represent spurious failures related to AP1.

Further, AP2 states that *the maximum roll angle allowed shall be between -33 and 33*. Similar to AP1, TurnKnob and APEng have appeared in most of the rules and invalid values of these variables play an important role in violating this requirement. However, we are not able to confirm six rules that either solely contain PitchWheel or their combinations with ALTMode (these rules are in bold and blue in Table 23).

Finally, rules in Table 24, are related to φ (i.e. AP3). One rule for AP3 is discussed in Section 2. PitchWheel and Throttle have appeared in most of the rules. Based on these rules, the requirement is failed if the airplane nose (i.e. PitchWheel) is in a downward position (i.e. negative value for PitchWheel) and the pilot does not provide the airplane with the enough throttle boost (i.e. Throttle). In addition to combination of PitchWheel and Throttle, the invalid values of PitchWheel and TurnKnob also identify spurious failures. For example, in rule 6 of Table 24, if the plane's nose is pointed downward and it is rotating sharply (high value for TurnKnob), it will drop altitude quickly and will therefore fail to satisfy the requirement (the value of 0 for ALTMode indicates that the airplane's altitude can be changed).

In addition, for all four subjects, we have examined how many rules can be obtained from the datasets generated by the preprocessing phase. our results indicate that none of the 46 rules confirmed as inducing spurious failures for autopilot could be obtained solely from the datasets generated by the preprocessing phase of SA (see Figure 1). Similarly, only two of the seven rules for NTSS could be obtained using the datasets from the preprocessing phase. Furthermore, from the preprocessing phase datasets alone, no additional candidate rules could be inferred for spurious failures in either NTSS or autopilot. These findings highlight the importance of utilizing surrogate-assisted test generation in order to obtain useful rules for spurious failures.

The answer to RQ4 is that failure models generated by the SA algorithm produce failure-inducing rules that indeed lead to spurious failures for the NTSS and autopilot case studies.

REFERENCES

[1] (Accessed: January 2023). *Autopilot online benchmark.* https://www.mathworks.com/matlabcentral/fileexchange/41490-autopilot-demo-for-arp4754a-do-178c-and-do-331?focused=6796756&tab=model

[2] (Accessed: January 2023). *CPS and NTSS requirements.* https://github.com/anonpaper23/testGenStrat/blob/main/Benchmark/Formalization/CPS_and_NTSS_Formalization.pdf

[3] (Accessed: January 2023). *Lockheed Martin.* https://www.lockheedmartin.com

[4] (Accessed: January 2023). *Logistic Regression.* http://faculty.cas.usf.edu/mbrannick/regression/Logistic.html

[5] (Accessed: January 2023). tc-cake. https://man7.org/linux/man-pages/man8/tc-cake.8.html

[6] Aitor Arrieta, Shuai Wang, Urtzi Markiegi, Ainhoa Arruabarrena, Leire Etxeberria, and Goiuria Sagardui. 2019. Pareto efficient multi-objective black-box test case selection for simulation-based testing. *Information and Software Technology* (2019).

[7] Federal Aviation Administration (FAA)/Aviation Supplies & Academics (ASA). 2009. *Advanced Avionics Handbook.* Aviation Supplies & Academics, Incorporated. https://books.google.lu/books?id=2xGuPwAACAAJ

[8] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars.. In *NDSS*.

[9] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. *ACM SIGPLAN Notices* 52, 6 (2017), 95–110.

[10] Raja Ben Abdessalem, Shiva Nejati, Lionel C Briand, and Thomas Stifter. 2016. Testing advanced driver assistance systems using multi-objective search and neural networks. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. 63–74.

[11] Bachir Bendrissou, Rahul Gopinath, and Andreas Zeller. 2022. "Synthesizing input grammars": a replication study. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 260–268.

[12] Devendra K Chaturvedi. 2017. *Modeling and simulation of systems using MATLAB® and Simulink®*. CRC press.

[13] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16 (2002), 321–357.

Table 21: List of inferred rules for NTSS in RQ4.

NTSS Rules
class7≥ 94%.thresh7∧ class5∈[71, 82]%.thresh5
class6∈[81, 91] %.thresh6∧ class7∈[69, 82]%.thresh7
class7∈[80, 91]%.thresh7∧ class5≥ 89%.thresh5
class7≥ 91%.thresh7∧ class2∈[60, 72]%.thresh2∧ class4∈[39, 51]%.thresh4
class7∈[79, 90]%.thresh7∧ class5∈[73, 80]%.thresh5
class6≥ 91%.thresh6∧ class7≥ 87%.thresh7
class5+ class6+ class7≥ 75%.threshold

Table 22: List of inferred rules for AP1 in RQ4.

AP1 Rules
APEng[0..300]=0∧TurnKnob[0..300] ≥4.0
APEng[0..300]=0∧ALTMode[0..300]=0∧TurnKnob[0..300]∈[17.5, 21.93]
ALTMode[0..300]=1∧TurnKnob[0..300]∈[17.5, 21.93]∧APEng[0..300]=0
APEng[0..300]=0∧TurnKnob[0..300]∈[7.61, 12.48]∧ALTMode[0..300]=0
APEng[0..300]=0∧ALTMode[0..300]=1∧TurnKnob[0..300]∈[8.74, 12.74]
APEng[0..300]=0∧ALTMode[0..300]=0∧TurnKnob[0..300]∈[21.75, 26.15]
APEng[0..300]=0∧ALTMode[0..300]=0∧TurnKnob[0..300]∈[35.97, 40.82]
APEng[0..300]=0∧ALTMode[0..300]=0∧TurnKnob[0..300]≥40.82
APEng[0..300]=0∧ALTMode[0..300]=0∧TurnKnob[0..300]∈[21.87, 26.33]
APEng[0..300]=0∧ALTMode[0..300]=0∧TurnKnob[0..300]∈[31.07, 35.97]
APEng[0..300]=0∧ALTMode[0..300]=0∧TurnKnob[0..300]∈[4.79, 9.13]
APEng[0..300]=0∧ALTMode[0..300]=1∧PitchWheel[0..300]∈[-16.64, -11.62]
APEng[0..300]=0∧ALTMode[0..300]=1∧PitchWheel[0..300]≥24.03
APEng[0..300]=0∧ALTMode[0..300]=1∧PitchWheel[0..300]∈[6.0, 12.06]
TurnKnob[0..300]∈[30.21, 35.08]∧PitchWheel[0..300]∈[-11.63, -5.54]
TurnKnob[0..300]∈[30.11, 34.8]∧PitchWheel[0..300]∈[-0.52, 5.96]
TurnKnob[0..300] ≥31.32

[14] William W Cohen. 1995. Fast effective rule induction. In *Machine learning proceedings 1995*. Elsevier, 115–123.

[15] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[16] Alan Díaz-Manríquez, Gregorio Toscano, Jose Hugo Barron-Zambrano, and Edgar Tello-Leal. 2016. A review of surrogate assisted multiobjective evolutionary algorithms. *Computational intelligence and neuroscience* 2016 (2016).

[17] Arkadiy Dushatskiy, Tanja Alderliesten, and Peter AN Bosman. 2021. A novel surrogate-assisted evolutionary algorithm applied to partition-based ensemble learning. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 583–591.

[18] Martina Friese, Thomas Bartz-Beielstein, and Michael Emmerich. 2016. Building ensembles of surrogates by optimal convex combination. *Bioinspired optimization methods and their applications* (2016), 131–143.

[19] Khouloud Gaaloul, Claudio Menghi, Shiva Nejati, Lionel C Briand, and David Wolfe. 2020. Mining assumptions for software components using machine learning. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 159–171.

[20] Rahul Gopinath, Alexander Kampmann, Nikolas Havrikov, Ezekiel O Soremekun, and Andreas Zeller. 2020. Abstracting failure-inducing inputs. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 237–248.

[21] Rahul Gopinath, Hamed Nemati, and Andreas Zeller. 2021. Input Algebras. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 699–710.

[22] Fitash Ul Haq, Donghwan Shin, and Lionel Briand. 2022. Efficient online testing for DNN-enabled systems using surrogate-assisted and many-objective optimization. In *Proceedings of the 44th International Conference on Software Engineering*. 811–822.

[23] Toke Høiland-Jørgensen, Dave Täht, and Jonathan Morton. 2018. Piece of CAKE: a comprehensive queue management solution for home gateways. In *2018 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*. IEEE, 37–42.

[24] Yaochu Jin. 2005. A comprehensive survey of fitness approximation in evolutionary computation. *Soft computing* 9, 1 (2005), 3–12.

[25] Yaochu Jin and Bernhard Sendhoff. 2002. Fitness Approximation In Evolutionary Computation-a Survey.. In *GECCO*, Vol. 2. 1105–12.

[26] Baharin Aliashrafi Jodat, Shiva Nejati, Mehrdad Sabetzadeh, and Patricio Saavedra. 2022. Learning Non-robustness using Simulation-based Testing: a Network Traffic-shaping Case Study. *arXiv preprint arXiv:2212.08726* (2022), to appear in 16th IEEE International Conference on Software Testing, Verification and Validation (ICST 2023).

[27] Alexander Kampmann, Nikolas Havrikov, Ezekiel O Soremekun, and Andreas Zeller. 2020. When does my program do this? learning circumstances of software behavior. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 1228–1239.

[28] Charaka Geethal Kapugama, Van-Thuan Pham, Aldeida Aleti, and Marcel Böhme. 2022. Human-in-the-loop oracle learning for semantic bugs in string processing programs. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 215–226.

[29] Fitsum Meshesha Kifetew, Roberto Tiella, and Paolo Tonella. 2017. Generating valid grammar-based test inputs by means of genetic programming and annotated grammars. *Empirical Software Engineering* 22, 2 (2017), 928–961.

[30] Neil Kulkarni, Caroline Lemieux, and Koushik Sen. 2021. Learning Highly Recursive Input Grammars. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 456–467.

[31] Jaekwon Lee, Seung Yeob Shin, Shiva Nejati, Lionel C Briand, and Yago Isasi Parache. 2022. Estimating Probabilistic Safe WCET Ranges of Real-Time Systems

Table 23: List of inferred rules for AP2 in RQ4.

AP2 Rules
APEng[0..300]=0∧ALTMode[600..900]=1∧ALTMode[300..600]=0∧TurnKnob[600..900]∈[26.67, 31.11]
TurnKnob[0..300] ≥ 30.46
APEng[0..300]=0∧TurnKnob[0..300] ≥ 4.0
TurnKnob[0..300] ≥ 40.41
TurnKnob[0..300] ≥ 30.04∧ALTMode[300..600]=1
ALTMode[300..600]=1∧TurnKnob[0..300]∈[9.52, 13.34]∧ALTMode[600..900]=0
APEng[0..300]=0∧ALTMode[300..600]=1∧TurnKnob[0..300]∈[30.76, 35.56]
ALTMode[0..300]=1∧TurnKnob[0..300]∈[13.34, 17.8]∧APEng[0..300]=0
PitchWheel[0..300]∈5.65-11.71∧PitchWheel[300..600]∈[-0.02, 6.26]
APEng[0..300]=0∧APEng[600..900]=1∧ALTMode[0..300]=1∧APEng[300..600]=1∧ALTMode[300..600]=0
APEng[0..300]=0∧ALTMode[600..900]=1∧APEng[300..600]=0∧ALTMode[300..600]=0∧APEng[600..900]=1
PitchWheel[600..900]∈[-24.07, -18.95]∧TurnKnob[600..900]∈[4.76, 9.01]
APEng[0..300]=0∧ALTMode[600..900]=1∧APEng[600..900]=1∧APEng[300..600]=0
APEng[0..300]=0∧ALTMode[300..600]=1∧APEng[300..600]=0∧APEng[600..900]=1
APEng[0..300]=0∧ALTMode[600..900]=0∧TurnKnob[300..600]∈[35.69, 40.28]
APEng[0..300]=0∧ALTMode[0..300]=1∧ALTMode[300..600]=0∧ALTMode[600..900]=0∧APEng[600..900]=1
APEng[0..300]=0∧ALTMode[0..300]=1∧TurnKnob[0..300]∈[8.19, 12.52]
APEng[0..300]=0∧ALTMode[0..300]=1∧ALTMode[600..900]=0∧ALTMode[300..600]=1∧APEng[600..900]=0
PitchWheel[300..600]∈[-24.81, -18.63]∧TurnKnob[600..900]∈[32.17, 35.67]
PitchWheel[0..300]∈[-24.07, -17.94]∧TurnKnob[600..900]≥40.51
TurnKnob[0..300]∈[26.01, 30.48]∧TurnKnob[600..900]∈[32.2, 36.96]
TurnKnob[600..900]≥41.09∧TurnKnob[300..600]∈[36.13, 40.39]
APEng[0..300]=0∧ALTMode[0..300]=1∧TurnKnob[0..300]∈[22.25, 26.43]
APEng[0..300]=0∧APEng[300..600]=1∧TurnKnob[0..300]∈[30.11, 34.8]

Table 24: List of inferred rules for AP3 in RQ4.

AP3 Rules
PitchWheel[0..300]≤-29.0∧ALTMode[300..600]=0.0∧PitchWheel[300..600]≤-29.0
PitchWheel[0..300]∈[-29.0, -27.93]∧Throttle[0..300]∈[0.13, 0.18]
PitchWheel[0..300]∈[-27.93, -22.54]∧ALTMode[0..300]=0.0∧Throttle[0..300]∈[0.015, 0.068]
PitchWheel[0..300]≤-29.0∧PitchWheel[300..600]≤-29.0
ALTMode[300..600]=0.0∧PitchWheel[300..600]∈[-29.0, -27.47]∧Throttle[300..600]∈[0.079, 0.14]
PitchWheel[0..300]∈[-22.11, -12.54]∧ALTMode[0..300]=0.0∧TurnKnob[0..300]∈[25.53, 30.64]
PitchWheel[0..300]∈[-22.11, -12.54]∧PitchWheel[300..600]∈[-10.61, -3.2]∧ALTMode[0..300]=1.0
PitchWheel[0..300]∈[-29.0, -25.49]∧ALTMode[0..300]=0.0∧ALTMode[300..600]=0.0∧PitchWheel[300..600]≤-29.0
PitchWheel[300..600]≤-29.0∧ALTMode[0..300]=0.0∧PitchWheel[0..300]≤-29.0
PitchWheel[0..300]∈[-27.7, -21.07]∧ALTMode[0..300]=0.0∧Throttle[0..300]∈[0.074, 0.13]
PitchWheel[0..300]∈[-28.43, -22.91]∧ALTMode[0..300]=0.0∧Throttle[0..300]∈[0.0073, 0.065]
ALTMode[300..600]=0.0∧PitchWheel[300..600]∈[-28.63, -23.33]∧Throttle[0..300]∈[0.065, 0.13]
PitchWheel[300..600]∈[-29.0, -27.24]∧ALTMode[0..300]=0.0∧ALTMode[300..600]=0.0∧Throttle[0..300]∈[0.075, 0.14]
PitchWheel[0..300]∈[-29.0, -27.6]∧Throttle[0..300]∈[0.13, 0.18]
PitchWheel[0..300] ≤ -28.0∧Throttle[0..300] ≤ 0.1

at Design Stages. *ACM Transactions on Software Engineering and Methodology* (2022).

[32] Jia Li, Shiva Nejati, and Mehrdad Sabetzadeh. 2022. Learning Self-Adaptations for IoT Networks: A Genetic Programming Approach. In *Proceedings of the 17th Symposium on Software Engineering for Adaptive and Self-Managing Systems* (Pittsburgh, Pennsylvania) *(SEAMS '22)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/3524844.3528053

[33] Sean Luke. 2013. *Essentials of Metaheuristics* (second ed.). Lulu. Available for free at http://cs.gmu.edu/~sean/book/metaheuristics/.

[34] Reza Matinnejad, Shiva Nejati, Lionel Briand, and Thomas Brcukmann. 2014. MiL testing of highly configurable continuous controllers: scalable search using surrogate models. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 163–174.

[35] Patrick E McKnight and Julius Najab. 2010. Mann-Whitney U Test. *The Corsini encyclopedia of psychology* (2010), 1–1.

[36] Claudio Menghi, Shiva Nejati, Lionel Briand, and Yago Isasi Parache. 2020. Approximation-refinement testing of compute-intensive cyber-physical models: An approach based on system identification. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 372–384.

[37] Claudio Menghi, Shiva Nejati, Khouloud Gaaloul, and Lionel C Briand. 2019. Generating automated and online test oracles for simulink models with continuous and uncertain behaviors. In *Proceedings of the 2019 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 27–38.

[38] Barton P Miller, Lars Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (1990), 32–44.

[39] Andrew Ng. 2018. Machine learning yearning. *Available: http://www.mlyearning.org/* (2018).

[40] Raphaël Ollando, Seung Yeob Shin, and Lionel C Briand. 2022. Learning Failure-Inducing Models for Testing Software-Defined Networks. *arXiv preprint arXiv:2210.15469* (2022).

[41] Ripon Patgiri, Hemanth Katari, Ronit Kumar, and Dheeraj Sharma. 2019. Empirical study on malicious URL detection using machine learning. In *International Conference on Distributed Computing and Internet Technology*. Springer, 380–388.

[42] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems* 25 (2012).

[43] Robert C Streijl, Stefan Winkler, and David S Hands. 2016. Mean opinion score (MOS) revisited: methods and applications, limitations and alternatives. *Multimedia Systems* 22, 2 (2016), 213–227.

[44] Hao Tong, Changwu Huang, Leandro L Minku, and Xin Yao. 2021. Surrogate models in evolutionary single-objective optimization: A new taxonomy and experimental study. *Information Sciences* 562 (2021), 414–437.

[45] Cumhur Erkan Tuncali, Georgios Fainekos, Danil Prokhorov, Hisahiro Ito, and James Kapinski. 2019. Requirements-driven test generation for autonomous vehicles with machine learning components. *IEEE Transactions on Intelligent Vehicles* 5, 2 (2019), 265–280.

[46] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[47] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 724–735.

[48] Ian H. Witten, Eibe Frank, and Mark A. Hall. 2011. *Data Mining: Practical Machine Learning Tools and Techniques* (3 ed.). Morgan Kaufmann, Amsterdam. http://www.sciencedirect.com/science/book/9780123748560

[49] Huanwei Xu, Xin Zhang, Hao Li, and Ge Xiang. 2021. An ensemble of adaptive surrogate models based on local error expectations. *Mathematical Problems in Engineering* 2021 (2021).