Machine-learning-Assisted Test Generation to Characterize Failures for Cyber-physical Systems

by

Abhishek Chandar

Thesis submitted to the University of Ottawa

in partial fulfillment of the requirements for the degree of

Master of Computer Science (Concentration in Applied Artificial Intelligence)

in

School of Electrical Engineering and Computer Science

University of Ottawa

Ottawa, Ontario, Canada

Abstract

With the advancements in Internet of Things (IoT) and innovations in the networking domain, Cyber-Physical Systems (CPS) are rapidly adopted in various domains from autonomous vehicles to manufacturing systems to improve the eficiency of the overall development of complex physical systems. CPS models allow an easy and cost-effective approach to alter the architecture of the system that yields optimal performance. This is especially crucial in the early stages of development of a physical system. Developing effective testing strategies to test CPS models is necessary to ensure that there are no defects during the execution of the system. Typically, a set of requirements are defined from the domain expertise to assert the system's behavior on different possible inputs. To effectively test CPS, a large number of test inputs is required to observe their performance on a variety of test inputs. But real-world CPS models are compute-intensive (i.e. takes a significant amount of time to execute the CPS for a given test input). Therefore, it is almost impossible to execute CPS models over a large number of test inputs. This leads to sub-optimal fixes based on the identified defects which may lead to costly issues at later stages of development.

In this thesis, we aim to improve the eficiency of existing search-based software testing approaches to test compute-intensive CPS by combining them with Machine Learning (ML). We call these ML-assisted test generation. In this work, we investigate two alternate ML-assisted test generation techniques: (1) surrogate-assisted and (2) ML-guided test generation, to eficiently test a given CPS model. Both the surrogate-assisted and ML-guided test generation can generate many test inputs. Therefore, we propose to build failure models that generate explainable rules on failure-inducing test inputs of the CPS

model. Surrogate-assisted test generation involves using ML as a replacement to CPS under test so that the fitness value of some test inputs are predicted rather than executing them using CPS. A large number of test inputs are generated by combining cheap surrogate predictions and compute-intensive execution of CPS model to find the labels of the test inputs. Specifically, we propose a new surrogate-assisted test generation technique that leverages multiple surrogate models simultaneously and dynamically selects the prediction from the most accurate label. Alternatively, ML-assisted test generation aims to estimate the boundary regions that separate test inputs that pass the requirements and test inputs that fail the requirements and subsequently guide the sampling of test inputs from these boundary regions. Further, the test data generated by the ML-assisted test generation techniques are used to infer two alternative failure models namely the Decision Rule Model (DRM) and Decision Tree Model (DTM) that characterizes the failure circumstances of the CPS model. We conduct an empirical evaluation of the accuracy of failure models inferred from test data generated by both ML-assisted test generation techniques. Using a total of 15 different functional requirements from 5 Simulink-based benchmarks CPS, we observed that the proposed dynamic surrogate-assisted test generation technique generates failure models with an average accuracy of 83% for DRM and 90% for DTM. The average accuracy of the dynamic surrogate-assisted technique has a 16.9% improvement in the average accuracy of DRM and a 7.1% improvement in the average accuracy of DTM compared to the random search baseline.

Acknowledgements

If at all I have contributed at some level towards conducting research at the intersection of Software Engineering and Machine Learning, I owe all the credits to my supervisor Prof. Mehrdad Sabetzadeh, co-supervisor Prof. Shiva Nejati and PhD candidate Baharin A. Jodat. This thesis would've not been possible if not for our combined discussions and their constructive criticisms of my work. They provided immense support over the past couple of years that not only shaped my research into what it is today but also shaped me as an individual. I also take this opportunity to thank my mom Jayashree, dad Chandar and my sister Sudharma. I am where I am only because of them. Last but never the least, I thank my friends Akshay Srinivasan, Nihal Antony, Surya Kiran Suresh and Tejas Atul Khare for creating a comfort zone where I could thrive the best.

Table of Contents

List of Tables

List of Figures

Abbreviations

C P S Cyber-Physical Systems ii, iii, v, xvi, 1–5, 7–13, 19, 26, 27, 29, 39, 50–54, 56, 59, 62, 98, 102, 104, 106–114

C P U Central Processing Unit 62

D R M Decision Rule Model 60, 71, 72, 76, 77, 80, 81, 85, 89, 93, 96, 97

D T M Decision Tree Model 60, 71, 72, 77, 80, 81, 85, 88, 89, 93, 96, 97

G P Gaussian Process 17

G P R Gaussian Process Regression 17, 23

I o T Internet of Things ii, 1

L R ML-guided test generation using Logistic Regression 50, 56, 58–60, 70–72, 76, 77, 80, 81, 85, 88, 89, 93, 96, 97

L S Least Squared 17

L S B LSBoost 16, 17

M A E Mean Absolute Error 35, 36

M L Machine Learning ii, iii, 22, 25, 26, 28, 30, 33, 35–37, 39, 40, 45–49, 54, 59, 62, 71, 80, 98, 100, 101, 104, 106

N N Neural Network 18, 19

R A M Random Access Memory 62

R F Random Forest 15, 16

R I P P E R Repeated Incremental Pruning to produce Error Reduction 20, 21, 46, 62

R S Random Search 27, 54, 56, 58–60, 70–72, 76, 77, 80, 81, 85, 88, 89, 93, 96, 97

R T Regression Tree 14, 56, 58–60, 70–72, 76, 77, 80, 81, 85, 88, 89, 93, 96, 97

S A Surrogate-assisted test generation 49–51, 53, 56, 58–60, 67–69, 72, 76, 77, 80, 81, 85, 88, 89, 93, 96, 97

S M O T E Synthetic minority over-sampling technique 26, 30, 32, 62

S V R Support Vector Regression 14

W C E T Worst-case Estimation Times 99

Chapter 1

Introduction

1.1 Context

With the recent advancements in IoT and networking domain, the application of CPS in various domains such as autonomous vehicles, manufacturing, networks, and even defense have contributed to improving the eficiency of existing physical systems significantly resulting in an improved quality of life for the stakeholders. CPS usually represents a dynamic mathematical model capturing a large number of interactions between software controllers and hardware systems.CPS industry primarily adopts Simulink, a widely known MATLAB tool for the development of CPS. A recent study shows that Simulink is adopted by more than 60% of engineers developing CPS [61]. Simulink provides a comprehensive set of functionalities to build, configure and test CPS.

Real-world CPS can be highly complex, involving numerous mathematical computa-

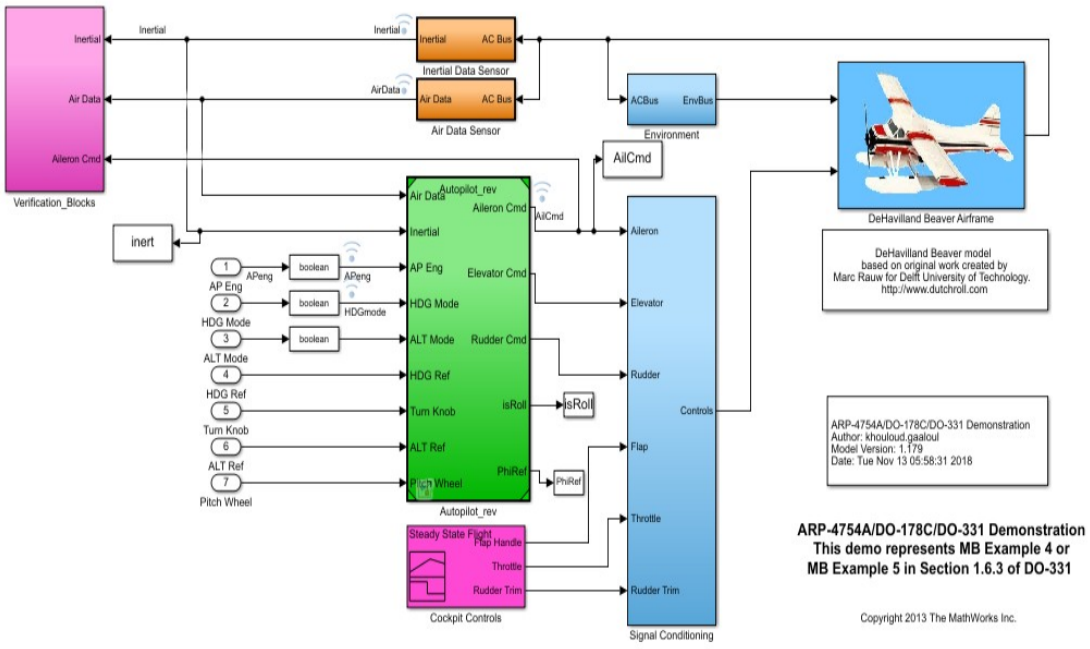

Figure 1.1: Example of CPS Autopilot

tions to execute the input signals and generate the output signals. A good example of a complex CPS is Autopilot which simulates a single-engine, high-wing, propeller-driven aircraft with all six degrees of freedom. The Autopilot CPS has 587 blocks, 623 connections, and 514 ports and is visually shown in Figure 1.1.

To ensure that there are no defects in the execution of the system, it is recommended that the CPS are tested eficiently and effectively to assert that the system satisfies its functional requirements. The primary goal of software testing approaches is to verify the correctness of the CPS against various functional requirements. To do so, testing CPS typically involves three phases. (1) Model-in-the-Loop (MiL): In this phase, a digital version CPS is modeled using programming tools such as Simulink for example to test the behavior of the system. Performing software testing on a simulator of CPS allows flexibility

in terms of changing the architecture of the CPS to achieve an optimized architecture to improve the eficiency of performing the mathematical computations and obtain desirable outputs for a given input. (2) Software-in-the-Loop (SiL): In this phase, the code representing CPS is implemented, and SiL is conducted to ensure that the code can execute on the target platform. (3) Hardware-in-the-Loop (HiL): At the final phase, the CPS is completely integrated with the complete control system and HiL testing verifies if the hardware implementation of CPS can operate with the control system without any failures.

In this thesis, we focus on the first phase of software testing i.e., MiL software testing that involves performing model-driven software testing of CPS. MiL testing is crucial for testing CPSs as identifying defects (i.e., failures) in the subsequent phases will lead to expensive fixes. There have been several research attempts previously to eficiently generate individual test cases. However, identifying individual test inputs that fail the system may not be always helpful in understanding the root cause of the failure. Recently, several approaches involve characterizing (i.e., explaining) failure circumstances of the system under test to better assist engineers in root cause analysis. Recent research on generating input grammars [36] [37] [8] [6] [57] and generalizing failure-inducing test inputs [35] [11] [28] [27] [35] are primarily employed to explain different failure circumstances of the system. Although these approaches are suitable for inputs of type string, the same approaches are not suitable for systems with numerical input. This is because the approaches dealing with string-based inputs have a binary verdict (i.e., pass or fail) as opposed to a quantitative measure that can measure the severity of the verdict. In this thesis, we adopt the latter to realize the performance of the CPS on different test inputs.

1.2 Challenges

To perform eficient software testing for CPS, we employ a data-driven test generation approach to learn failure models. Failure models involve learning a set of conditions explaining the failure circumstances and essentially classifying test inputs that cause failure and the test inputs that do not. Some of the applications of failure models in the context of software testing are: (1) Debugging a CPS module, (2) Identifying invalid inputs that could falsely degrade the performance of the system, (3) Generating additional failure test inputs similar to the ones learned by the failure model and finally, (4) Identifying the root cause of the failure circumstances of the CPS. To generate test data to train failure models, two alternate approaches can be applied:(1) Explorative search and (2) Exploitative search [40]. Both these approaches generate test inputs by sampling over a search space. A search space covers all test inputs that are in the operating range of CPS. Explorative search methods samples test inputs from all parts of the search space to increase the coverage of searching over input search space for failure-inducing test inputs. Alternatively, the exploitative search method samples specific areas of interest from the input search space that are more informative in differentiating the test inputs that satisfy and fail the requirements. The challenges addressed in performing eficient model-driven software testing for CPS are as follows:

- The first challenge deals with performing the explorative search. The execution of CPS is computationally expensive and time-consuming. For example, based on our experiments, the CPS Autopilot takes about 0.5 minutes to execute a test input for all the requirements pertaining to Autopilot. CPS are designed to conduct individual

simulations independently but are not suitable for performing a large number of simulations for software testing. Therefore, the first challenge addressed in this thesis is to generate a large number of tests within a given time budget. In the case of compute-intensive models under test, generating a lot of tests may even be infeasible.

- The second challenge deals with performing the exploitative search. The challenge with performing an exploitative search is that an effective guiding mechanism is required to exploit a particular area of interest in the input search space.

- Finally, the third challenge revolves around identifying the patterns of failure circumstances at a system level. Analyzing the failure-inducing test inputs individually may result in a sub-optimal solution that provides a fix specific to only a subset of failure-inducing test inputs. A range of similar failure-inducing test inputs may not satisfy the requirements of CPS. Therefore, the third challenge is to incorporate a module that characterizes the failure at a system level into the software testing framework. This allows the engineers to see the big picture in identifying the root cause of the problem and provide a solution that is closer to the global optimal solution.

1.3 Research Contribution

The following research contributions are made through this research:

- We present a dynamic surrogate-assisted test generation algorithm that dynamically uses multiple surrogate models simultaneously during the search, and selects the most

accurate prediction model. We evaluated seven different surrogate models identified in the literature [29] [54] [20] [21] and based on our analysis, we found that compared to using surrogate models individually for prediction, the proposed dynamic surrogate-assisted algorithm resulted in the most optimal trade-off between accuracy and eficiency by generating datasets that are at least 33% larger and 28% more accurate (RQ1 in Section 4).

- We demonstrate that the two alternative ML-guided test generation algorithms proposed in this research are more eficient in guiding the sampling towards the areas of search space that differentiate the test inputs that pass the requirement and the test inputs that fail the requirement when compared to the random search algorithm. (RQ2 in Section 4).

- We perform a comparative analysis of the accuracy of failure models trained using our dynamic surrogate-assisted algorithm along with two ML-guided test generation algorithms (discussed in Chapter 3) and a random search algorithm (discussed in Chapter 2) as a baseline. (RQ3 in Section 4).

1.4 Organization

The remainder of this thesis is organized as follows. Chapter 2 covers the background concepts that help the reader to understand the domain of software testing in the context of cyber-physical systems and Machine learning. Chapter 3 discusses the overall framework of the ML-guided test generation framework proposed in this thesis. Specifically, we intro-

duce two broad test generation techniques namely Surrogate-assisted test generation and ML-guided test generation. Chapter 4 introduces the evaluation strategy of this research discussing the research questions, evaluation setup, study subjects, and evaluation metrics to evaluate the frameworks proposed in this thesis. Chapter 5 presents a comprehensive set of results and analyses to answer the research questions. Chapter 6 covers the related works to understand the latest advancements in the field of software testing for CPS. Finally, Chapter 7 presents the conclusion, future directions, and final thoughts related to this research.

Chapter 2

Background

This chapter covers all the foundational backgrounds related to machine learning and automated software testing of CPS.

2.1 System modeling for CPS

CPS [7] [52] [38] is a combination of networking, computation, and physical components into a single complete system of components that interacts with each other to execute necessary processes. A typical CPS is represented by a large number of computational components that interact with each other to process the input. Therefore, it is important to ensure that CPS is robust and can adapt to overcome system failures. This calls for advanced software testing methods to validate and verify individual components of CPS and the system as a whole. CPS are characterized by complex dynamic interactions that exhibit

time-varying changes based on the modeling paradigm. The changes can be continuous or discrete.

Model-based design [24] [34] is an advanced design technique to represent CPS digital twin as a center of the development process and allows to conduct verification of CPS eficiently. MATLAB-based Simulink is a powerful tool used to design, develop and interact with CPS.

2.2 MATLAB-based Simulink

Simulink is a block diagram environment used to design systems with multidomain models, simulate before moving to hardware, and deploy without writing code [1]. Simulink is a MATLAB toolbox that provides data-flow-driven block diagrams to simulate a physical system (for example, CPS) before developing the hardware. One of the common applications of Simulink is to perform simulation-based software testing with CPS under test to identify and eliminates errors in the early stages of development avoiding costly fixes in later stages.

Simulink is a programming language used for the modeling of dynamic numerical systems. It facilitates the development of digital twins of advanced CPS in the automotive, telecommunication, aerospace, and so on. Simulink consists of a library to develop the building blocks of a CPS namely blocks, input/output ports, and the connections between different blocks. Blocks define the mathematical operations of a system. Ports specify any signal data that pass through blocks. Finally, connections establish the flow of data be-

tween different ports in a block. CPS are developed as a set of subsystems each accepting a specific number of input signals and generating a fixed number of output signals.

To execute a CPS S, the input signals are defined over some time steps and then the output signals are computed for each time step of the inputs. Execution of the simulation is defined by $H(\bar{u}, S) = \bar{y}$. Here, S represents CPS under test, $\bar{u} = \{u_1, u_2, \ldots, u_m\}$ represents inputs signals and $\bar{y} = \{y_1, y_2, ..., y_n\}$ represents a set of output signals. Specifically, the input signals and output signals are generated over a time domain which is defined as a non-singular bounded interval $T = [0, b]$ of R where $R = \{R_1, \ldots, R_n\}$. Here, $\{R_1, \ldots, R_n\}$ are the range of the input signals $\{u_1, u_2, \ldots, u_m\}$.

2.3 Model-Based Verification of CPS

2.3.1 Model-Based Testing

There are two main types of model-based testing approaches: (1) Ofline and (2) Online.

Ofline model-based testing: Ofline testing performs the generation of multiple test inputs for a given time budget completely followed by the execution of the generated test inputs. In other words, ofline model-based testing creates an apriori of test inputs first and then executes them one by one.

Online model-based testing: Online testing couples the generation and execution of test inputs together. In other words, for a given test input, the approach generates a test input and executes the test input before sequentially moving to the next test input until

the execution time budget is reached. One of the main advantages of online model-based testing over ofline model-based testing is that the former allows the scope to refine the test input search space to effectively search in specific areas of interest. We have explored such techniques in this thesis which is discussed elaborately in Chapter 3. Finally, all the algorithms and baselines discussed in this thesis apply Online model-based testing.

2.3.2 Fitness Functions

Every CPS under test can have one or more requirements that must be tested to ensure that the system operates as intended. A fitness function is a form of test oracle that determines whether the CPS under test has passed or failed the requirement. In the case of software testing for the CPS, the fitness function maps the output signals of the simulator for a given test input to a quantitative value. Although, in our work, we focus only on testing the functional requirements of the CPS, it is to be noted that the non-functional requirements also need to be considered for testing to identify potential failures. For each functional requirement, a fitness function is developed based on previous work by Menghi et al. [26] where the fitness functions are quantitative. Specifically, the fitness function returns a fitness value that determines the degree of the CPS under test either passes the requirement or fails the requirement. All the fitness functions used in this thesis are designed such that a fitness value greater or equal to 0 refers to the system passing the requirement under the test. If the fitness value is less than 0, the system fails the requirement under the test.

Algorithm 2.1 Random Search - Baseline.

Input. S: The Simulation System
Input R = $\{R_1, \ldots, R_n\}$: Range for input variables v_1 to v_n
Input F: Fitness function
Input Budget: Maximum number of fitness evaluations
Output. DS: A dataset to train failure models

```
1: function DS=RANDSEARCH(S, R, Budget)
2:     while Budget do
3:     p=GEN( R);                                  ▯ Generate
4:     f=SIM( S, p);                               ▯ Simulate a test input
5:     l=Label(f,F);                               ▯ Label a test input as pass and fail
6:     DS = DS ▯{f, l, p}
7:     end while
8:     return DS
9: end function
```

2.3.3 Search-based software testing

Considering the larger input feature space for CPS, search algorithms are commonly applied to perform software testing of CPS and identify test inputs that violate (i.e., fail) a requirement. To perform software testing, the search algorithms sample the input search space and choose a test input for evaluation using the CPS under test. To evaluate if a test input is passing or failing a requirement, fitness functions are defined for each requirement. Random Search (RS) [12] [50] [46] is one of the most common search approaches implemented to sample the input search space. In this thesis, RS is used as a baseline for the ML-guided frameworks elaborately discussed in Chapter 3.

2.3.3.1 Random Search

Algorithm 2.1 shows a step-by-step approach to performing automated software testing for the CPS under test. RS takes as input the simulator S, a range for each input variable R defining a search space and fitness function F. The algorithm begins by randomly generating a test input p from R (Line 3). Then, p is passed to the simulator S to perform an execution of S to obtain a fitness value f (Line 4). We then find the label l of fitness value f based on the fitness function F (Line 5). Recall that f quantitatively defines how well the CPS performed for a given input p. According to all the fitness functions defined for requirement subjects used in this thesis for evaluation, if the fitness value f is greater than or equal to 0, then the simulator passed the requirement under the test. If the fitness value is less than 0, then the simulator has failed the requirement. Finally, the dataset is updated with the test input p, fitness value f, and label l (Line 6). Line 3 to 6 is iteratively performed until the time budget for executing the CPS under test has been reached. RS samples test inputs in a completely random manner. Therefore, applying RS for software testing, especially for computationally expensive systems is not eficient. This is one of the primary motivations for this research in exploring ML-assisted test generation techniques to improve the eficiency of software testing techniques for testing computationally expensive CPS.

2.3.4 Supervised Machine Learning - Regression Techniques

2.3.4.1 Support Vector Regression

Support Vector Regression (SVR) [22] is a supervised machine learning algorithm that is used for regression tasks. The algorithm is similar to its classification alternative (i.e., Support Vector Machine) used for classification since both algorithms attempt to fit a hyperplane for the training data. SVR is fit on the training data by passing a set of input vectors and its corresponding target feature. As the model attempts to solve a regression problem, the goal of SVR is to find a function $f(x)$ where x is the input such that the predicted value of the target feature does not deviate from the actual value of target feature by any more than a threshold value ϵ for each data point. SVR allows the train data to be projected on a higher dimension using the kernel parameter which is defined as the function that maps lower-dimensional data into higher-dimensional space. A linear kernel is used when the size of the training data is high. On the other hand, if the dataset is non-linear, projecting the training data in a higher dimensional representation may help in improving the accuracy of the prediction. SVR is implemented using the MATLAB function fitrsvm.

2.3.4.2 Regression Tree

Regression Tree (RT) [14] is a type of decision tree in which the target feature can be of the continuous type as opposed to categorical labels. At a high level, the regression tree attempts to split the dataset into multiple subsets. Specifically, a greedy recursive approach is taken to divide the dataset into subsets in which the partitions performed earlier in the

iterations will not be altered based on later partitions. In each iteration, the best split that divides the data into two distinct regions is identified by the split value of the target feature such that the overall sum of squares error of every input feature is minimized. This process is iteratively performed until the stopping criteria are reached. Generally, deeper regression trees tend to be highly specific to the training dataset. Therefore, overfitting is one of the most common issues of regression trees. To overcome this issue, hyperparameter tuning is performed that finds the optimal parameters of the regression tree reducing the chances of the trained regression tree overfit on the train data. The parameters of the regression tree that were tuned using Bayesian Search Optimization are MinLeafSize defined as the minimum number of leaf node observations and MaxNumSplits defined as the Maximum number of decision splits to divide the dataset. The regression tree is implemented using the MATLAB function fitrtree.

2.3.4.3 Tree-based Bagging technique

Random Forest (RF) [15] is a supervised learning algorithm that uses ensemble techniques under the hood to predict classification and regression tasks. More specifically, the ensemble technique applied is called bagging in which several small subsets of training data are created and multiple weak models are trained individually on them. Formally, a weak model is defined as those machine-learning models that perform slightly better than random guessing. Further, the predictions from each weak model are averaged which usually results in a more accurate prediction when compared to individual predictions. The configuration settings related to RF specifically in Matlab implementation are MaxNumSplits, MinLeafSize, and NumLearningCycles. MaxNumSplits defines the maximum num-

ber of children nodes that are allowed for a given node in the tree. MinLeafSize is the minimum number of observations in the training data that is satisfied by a leaf node. During the training, if the MinLeafSize is exceeded in any of the leaf nodes, the training process is stopped. A high MinLeafSize value will create a RF model that is too deep resulting in a model that is overfitting. On the other hand, a low MinLeafSize value will lead to a shallow tree that is not able to understand the latent patterns in the training data. In other words, the model results in underfitting. Therefore, setting appropriate values for MinLeafSize and MaxNumSplits is essential to identify the optimal tree for a given dataset. Further, RF trains multiple independent weak models and performs the overall prediction by averaging the individual prediction from each weak model. This is one of the primary reasons for the RF algorithm's ability to handle overfitting well. RF is implemented using the MATLAB function fitrensemble.

2.3.4.4 Tree-based Boosting technique

LSBoost (LSB) [13] is a gradient-boosting machine-learning algorithm in which a weak model is initialized and gradually improved over several iterations. In other words, Boosting techniques primarily apply a type of machine-learning algorithm such as a decision tree, or regression tree, and then improve it until the stopping conditions are met. The stopping conditions are usually predefined by setting the hyperparameters. For example, as explained in the previous subsection, setting parameters such as MinLeafSize determines when the training can be terminated. In boosting algorithms used for regression tasks, the objective function used to improve the weak learners captures the difference between the predicted value and the actual ground truth value. The most common objective function

(also commonly referred to as loss function) is Least Squared (LS) error. Our implementation also uses LS as the objective function and hence the name LSBoost. In each iteration, the algorithm fits a new weak learner such that the Least-Squared error is lower than the cumulative predictions of all the previous weak models thereby improving over several iterations. Unlike the Bagging technique like Random Forest where the overall prediction is obtained by averaging the individual predictions with equal weights, Boosting techniques average the individual predictions such that more weight is given to the weak learners that have high performance which is defined by a lower loss function value. LSB is implemented using the MATLAB function fitrensemble.

2.3.4.5 Gaussian Process Model

Gaussian Process (GP) [48] is a probabilistic model used to perform supervised regression and classification tasks. In this thesis, GP is used to predict a continuous target feature. Therefore, it is used to perform a regression task. Gaussian Process Regression (GPR) is a nonparametric model that allows calculating probability distribution for multiple functions that is permissible for a given data. To make a prediction, Gaussian Process Regression calculates the covariance (σ) between the input features. Further, the covariance values between all possible pairs of input features are used to estimate a probability distribution of the target feature for a given train data. Gaussian Process Regression assumes that the target feature is Gaussian in nature. GPR is especially effective in mapping unknown non-linear relationships between input features and target output features. Further, GPR allows the data to be projected in both linear and non-linear kernels. In this thesis, GPR with linear and non-linear kernels was implemented using MATLAB function fitrgp.

2.3.4.6 Neural Network

Neural Network (NN) [19] is a mathematical model that is inspired by the biological structure of neural networks in the human brain. Depending upon the kind of prediction task, neural networks can be suited for both regression and classification tasks. Our work requires us to predict real values and therefore we implemented the neural network to predict real values. Additionally, since we had labeled the dataset, the neural network was trained in a supervised manner. The structure of the neural networks can be broadly discussed using three types of layers where each layer has several neurons. The first layer is the input layer in which the inputs are initially fed into the NN for prediction. The second set of layers is the hidden layer. There can be one or more than one hidden layer. The number of neurons in the hidden layers and the number of hidden layers itself can be experimented with to find the optimal neural network that results in accurate predictions. The last layer is called the output layer which outputs the final prediction. NN is incorporated for experimentation when there is a lot of data with little knowledge about the underlying patterns in the data. Most of the machine-learning models assume the data in which it is being used for prediction. In the case of neural networks, there is no assumption about the characteristics of the data being made.

In NN, each neuron in a layer has a weight associated with it. The input values multiplied by the respective weights of the neurons are passed from the input layer to the output layer through the hidden layers. At the output layer, the predicted output is compared with the actual output to compute a loss. This loss is back-propagated to recompute the weights of all the neurons in each layer. In other words, the neural network

is fine-tuned based on the error between the actual output and predicted output in each iteration. Previous work in predicting the fitness score of CPS in the context of automated software verification has shown that a neural network with two hidden layers with each layer having (4/3)⍰N nodes where N is the number of inputs of a CPS has shown to result in accurate predictions [10]. NN is implemented using the MATLAB function fitrnet.

2.3.5 Supervised Machine Learning - Classification Techniques

2.3.5.1 Decision Trees

Decision Trees [49] [51] are a well-known machine learning model primarily used for classification tasks when the target feature is categorical and for regression problems when the target feature is continuous. As the name suggests, decision trees build a tree structure based on a divide-and-conquer approach to divide the dataset. The decision tree is defined by the root node, internal nodes, and the leaves of the tree. Each internal node is labeled with input features. The edges connecting a parent node with its child node are labeled with a condition on an input feature. Each leaf on the decision tree represents the predicted class of the target feature. Multiple metrics have been adopted to evaluate the best feature to split the dataset in which iteration of building the decision tree. Some examples of metrics that are used to measure the eficiency of the split are Information gain and Gini Impurity.

2.3.5.2 Decision Rules

Decision Rules [18] is a set of IF-THEN conditions that are learned from the input features from the training data to perform a prediction. They are one of the most commonly used interpretable models apart from Decision Trees to better understand the reason behind the prediction. A prediction is performed with the help of a single decision rule or a combination of multiple decision rules. Many algorithms adopt decision rules to effectively perform predictions. Some of the approaches are (1) OneR and (2) Sequential covering.

OneR: It is one of the simplest forms of decision rules. The algorithm first identifies the number of times each value for every feature in case of each class has appeared. The algorithm then chooses the value with the maximum number of correct classifications for every feature. Then, a rule is created for every feature and the conjunction of all the rules is considered for prediction.

Sequential covering: Sequential covering is another decision rule approach that creates a set of rules that covers the training dataset completely. Repeated Incremental Pruning to produce Error Reduction (RIPPER) [18] is a robust well-known implementation of sequential covering decision rules. In this thesis, RIPPER is implemented representing decision rules to build failure models. At a high level, RIPPER first identifies an initial rule that applies to a small subset of data points. The algorithm then removes the data points covered by the initial rule from the dataset. A new rule is identified that represents the new set of data points. For every rule that is generated, RIPPER applied rule pruning which is a postprocessing step to prune any redundant rules that do not help in improving the overall performance of the decision rules. This process is performed iteratively until

all the data points from the training data are covered by the set of rules or the stopping criteria are satisfied.

In this thesis, Decision trees and RIPPER are the two machine learning models that are used to build failure models as elaborately described in Chapter 3.

2.3.6 Hyperparameter Tuning

Hyperparameters of machine learning models are defined as the parameters that can be tweaked before training the model on a dataset. It can be considered as the configuration settings of machine learning to optimize the training and inference process of the machine learning model. Tuning parameters is therefore essential for optimally training a machine learning model and in turn avoiding negative phenomena such as overfitting.

Hyperparameter optimization is defined as the process to search for optimal hyperparameter values that lead to the maximization of the performance metric used for evaluation. Some of the most common ways to perform hyperparameter optimization are: (1) Manual Tuning (2) Grid Search, (3) Random Search, and (4) Bayesian Search Optimization. These approaches are discussed below.

- Manual Tuning: Manual search involves manually setting a set of hyperparameter settings and observing the best possible hyperparameter configuration concerning the train data.
- Grid Search: Grid Search takes the search space with finite values for each hyperparameter as input. The algorithm exhaustively searches all possible hyperparameter

configurations, train the machine learning model, and evaluates the model using performance metrics.

- Random Search: As the name suggests, the approach randomly selects a set of hyperparameter configurations and trains the ML model to evaluate the performance metric on the evaluation dataset. In this case, since each hyperparameter setting is randomly selected from the search space, we need to define a search space of hyperparameter value in which a random search can be performed. Further, a maximum number of iterations that is conducted before returning the optimal hyperparameter setting is set before executing the tuning process.

- Bayesian Search: Hyperparameter tuning is a type of optimization problem. Let $f(x)$ be the black box objective function for tuning hyperparameters. If $f(x)$ is cheap to execute, Grid Search and Random Search can be performed to exhaustively search through multiple hyperparameter configuration possibilities. But if evaluating the function $f(x)$ is expensive which is the case for many Machine-learning and Deep Learning models, then we cannot afford to perform a large number of evaluations to arrive at the optimal hyperparameter configuration. Therefore, an eficient approach needs to be adopted such that the optimal hyperparameters are identified in a minimum number of evaluations. Bayesian Optimization can achieve this by using prior information from previous iterations of evaluations to take an informed decision in choosing the next set of hyperparameters. In this thesis, we apply Bayesian search optimization to tune the hyperparameters of all the machine learning models used for regression and classification tasks. More details on Bayesian Search for

hyperparameter tuning are discussed below:

2.3.7 Bayesian Search Optimization

As discussed above, the objective function $f(x)$ to find the optimal hyperparameter setting can tend to be complex. At a high level, the idea of Bayesian search optimization is to create a surrogate model such as GPR that approximates $f(x)$ by evaluating fewer points. In simple terms, Bayesian search optimization considers the previous iterations of evaluating hyperparameter settings to select the next hyperparameter setting of evaluation thereby reducing the number of evaluations needed to find the global maxima. Unlike other hyperparameter tuning methods such as Grid Search or Random Search, the hyperparameter settings are selected based on statistically guided knowledge. This is the main reason for Bayesian Search Optimization performs better than both Grid search and Random search [25]. Bayesian Search optimization involves the following steps:

- (1) An initial set of sample points are created. Here, each sample point refers to the evaluation of function $f(x)$ where x is a hyperparameter configuration selected for evaluation.
- (2) A black box surrogate model is first created based on the initial set of sample points. This black box model can be Gaussian Process regression for example.
- (3) An acquisition function (α_x) is then selected. Acquisition functions are a form of metric based on which the next sample point for evaluation is selected. Specifically, the acquisition function controls the explorative-exploitative tradeoff in selecting the

next sample point. The next query point is selected such that the performance metrics of the machine learning model are maximized and the uncertainty of the point is high. The acquisition function used to tune the hyperparameters in this thesis is called Expected Improvement (EI). The intuition of EI is to maximize the expected degree of improvement over the sample point with the current maximum ($f(x^+)$). Since $f(x)$ is unknown and the goal is to find x such that the improvement is maximized, the expected improvement is calculated as follows:

$$EI_n(x) = E_n[[f(x) - f(x^+)]] \tag{2.1}$$

In Equation 2.1, E_n is the expectation calculated for a given distribution $f(x)$ representing all the evaluations of points x1, ...x_n.

- (4) The acquisition functions are iteratively optimized to choose the next sample point.

$$x_{next} = \text{argmax}(\alpha_x) \tag{2.2}$$

The next point x_{next} is selected such that the acquisition function is maximized. Finally, x_{next} is evaluated and the black box model is updated with the new evaluation. Steps (2) to (4) are performed iteratively until convergence is reached or the maximum iterations have been reached.

2.3.8 Imbalance Handling

A dataset is imbalanced when one or more classes have more representation than the other classes. For example, in a binary classification problem, if 50% of the dataset contains the positive class and the remaining 50% of the dataset contains the negative class, then the dataset is considered a balanced dataset. On the other hand, a dataset is said to be heavily imbalanced if 90% of the dataset contains a positive class and the remaining 10% of the dataset represents a negative class. Data imbalance can hurt the performance of the ML model. More specifically, as the data imbalance increases, the ML may tend to prioritize the majority class as opposed to the minority class. To avoid the ML models showing bias towards the majority class, data imbalance should be handled. There are two main methods to handle data imbalance: (1) Undersampling and (2) Oversampling.

Undersampling is a technique to rebalance the distribution of positive and negative classes by reducing the number of majority class data points. This in turn reduces the skewness in the dataset. This technique is applied when there is a large amount of data. Oversampling is another technique used to alter the distribution of positive and negative classes to reduce the skewness between the classes. In this approach, duplicate data points of the minority class data points are created to increase the representation of the minority class. Since it may not be practical to collect a large amount of test data to perform undersampling, in our framework, we resort to oversampling to handle data imbalance. Oversampling can be performed in two ways: (1) Naive random oversampling in which the duplicate data points are created randomly from the minority class (2) SMOTE which performs a systematic approach of creating synthetic data points near the minority class

data points in the search space. Specifically, Synthetic minority over-sampling technique (SMOTE) oversamples 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 neighbors [17].

Since SMOTE takes an informed decision to generate synthetic data points representing the minority class when compared to the naive random oversampling technique, we apply SMOTE to perform handle data imbalance in our algorithms.

2.4 Overview of related works

This research work primarily focuses on three main aspects at the intersection of software verification and Machine Learning. They are (1) ML for Search-based software testing, (2) Surrogate-testing, and (3) Building failure models. Various ML models have been adopted to improve the eficiency of Search-based software testing. Previous works such as Lee et al. [39] and Feldt et al. [23] follow a well-known strategy to adopt ML for search-based software testing. First, an initial set of test inputs are generated using search algorithms and then these initial test inputs are used to train ML models to assist the search algorithms. Surrogate testing is another well-known area of research in software verification where a surrogate is created that acts as a replacement to the actual system under test. Previous research works such as Dushatskiy et al. [21], Haq et al. [29], Abdessalem et al. [10], Beglerovic et al. [9] have adopted ML models as surrogates to CPS under test. Finally, building failure models to characterize the failure circumstances of a system is an active area of research. Previously, failure models were developed for software programs

with string-based inputs. Previous works such as Kampmann et al. [35], and Gopinath et al. [27] are some related works to this thesis. Our research focuses on building failure models to characterize failure-inducing test inputs of CPS with numerical test inputs. A detailed description of the related works for this thesis has been presented in Chapter 6.

2.5 Summary

In this chapter, we discussed the necessary background concepts related to this thesis. We first discussed how system modeling is performed in the context of CPS followed by understanding Simulink and how it is used to develop CPS. We then explored the definition of model-based software verification for CPS. We defined two types of model-based testing namely Ofline and Online model-based testing. Further, we defined fitness functions that are used to evaluate the performance of the simulator under test for a given test input. We briefly discussed a high-level workflow of search-based software testing which is commonly adopted in testing CPS especially due to its large input search space. We then established a common search-based software testing technique called Random Search (RS) which is considered the baseline for the ML-assisted frameworks for software testing proposed in this thesis. We then covered a high-level description of various supervised machine-learning techniques adopted in this work for both regression and classification tasks. Finally, the concept of hyperparameter tuning and imbalance handling is established which is predominantly used in the algorithms described in Chapter 3.

Chapter 3

Approach

This section discusses the approach of the proposed algorithms. The framework is elaborated in the following sections as follows. Section 3.1 discusses the inputs and outputs of the framework. Section 3.2 and Section 3.3 describes the overall workflow of both surrogate-assisted test generation and ML-guided test generation frameworks respectively.

3.1 Inputs and Outputs

Figure 3.1 shows the framework to perform the proposed ML-assisted test generation for buildings failure models. The inputs to the framework are: (1) an executable cyber-physical system or simulator under test S, (2) the input search space representation R for S, and (3) a quantitative fitness function F for each requirement of S. The following are the two assumptions that are made about the input search space (R) and the fitness function (F):

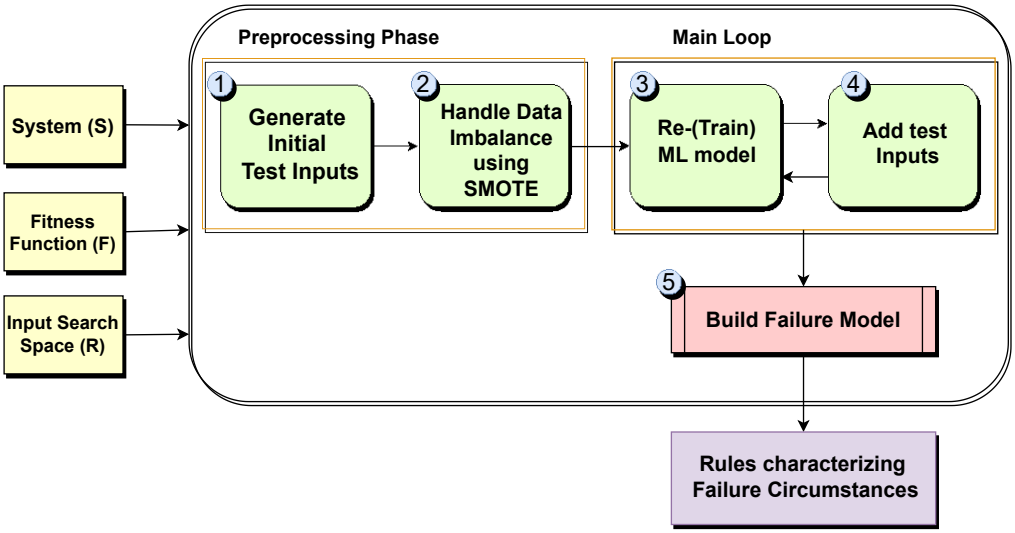

Figure 3.1: Conceptual diagram showing the framework for generation test data to build failure models. The framework encapsulates two Machine-learning assisted test generation approaches i.e.„ Surrogate-assisted and ML-guided test generation. Both algorithms differ in the main loop.

(Assumption 1): We assume that the system inputs are variables of type real values. For each input variable, the range of the values the variable can take is defined by an upper and lower bound.

(Assumption 2): For each requirement of S, we have a fitness function F based on which pass/fail labeling can be performed for any given test input.

We assume that the range of fitness value obtained by F is an interval $[-a, b]$ of R. For any given test t, $F(t) \geq 0$ 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. Both Assumption 1 and Assumption 2 are common for CPS expressed in Simulink [4, 16, 44, 55], automated driving systems [29], and are valid for all the case studies we use in our evaluation (see Section 4.3).

Table 3.1: An example CSV file structure of dataset generated by the proposed algorithms.

FitnessValue	Label	Type	TestInput
-0.89	FAIL	SIM	[1.22,-2.60 ,-13.66]
5.06	PASS	PRED	[7.06, 8.61, -17.11]

We examine two different ML-assisted test-generation approaches for building failure models: (1) surrogate-assisted and (2) ML-guided. Both approaches can be captured using the framework in Figure 3.1. There are 5 main steps in the framework. Step 1: The preprocessing phase generates a set of test inputs labeled as pass or fail based on the fitness values. Step 2: This step involves balancing the dataset by oversampling the minority class using SMOTE. Step 3: The dataset generated by the preprocessing phase is used to perform supervised learning of a ML model in the main loop. When the framework is applied for surrogate-assisted test generation, the model predicts fitness values for the generated test inputs. When the framework is applied for ML-guided test generation, the model guides the sampling of test inputs. Step 4: The main loop extends the dataset from the preprocessing phase while refining the machine learning model based on newly generated tests. Step 5: After the main loop terminates, the framework uses the extended dataset to train a failure model such as Decision Rules or Decision Trees. Failure models are then used to characterize the failure circumstances of the system in the form of rules explaining the failure. The structure of the dataset generated by the algorithms discussed in this thesis is shown as an example in Table 3.1. The requirement subjects used for experimentation are mentioned in Section 4.3. In the remainder of this section, we detail each step of the framework in Figure 3.1.

Algorithm 3.1 Preprocessing Phase

Input. S: The Simulation System
Input R = $\{R_1, \ldots, R_n\}$: Range for input variables v_1 to v_n
Input F: Fitness function
Input Budget: Maximum number of fitness evaluations
Output. DS_{bal}: A balanced dataset with initial set of test inputs

```
1: function DS=PREPROCESS(S, R, F, Budget)
2:     while Budget/4 do
3:     p=GEN(R);                          □ Generate a test input randomly
4:     f=SIM(S,p);                                □ Simulate a test input
5:     l=Label(f,F);                      □ Label test input as Pass or Fail
6:     type = "SIM";
7:     DS = DS □{f, l, type, p};
8:     end while
9:     DS_bal = HandleImbalance(DS)
10:    while |DS_bal| ≤ Budget/2 do
11:    p=GEN(R);                          □ Generate a test input randomly
12:    f=SIM(S,p);                                □ Simulate a test input
13:    l=Label(f,F);                      □ Label test input as Pass or Fail
14:    type = "SIM";
15:    DS_bal = DS_bal □{f, l, type, p};
16:    end while
17:    return DS_bal
18: end function
```

3.2 Preprocessing Phase

Algorithm 3.1 refers to the preprocessing phase of the framework depicted in Figure 3.1. This phase generates an initial dataset utilizing a total of 50% of the execution budget. The step-by-step process is explained as follows - The algorithm first begins by generating 25% of the budgeted test inputs (i.e., (Budget/4)) and computes a fitness value f for test input p by executing S (line 3 to 4). We then determine to label l for p based on the fitness function

F (lines 5). The variable type ∈ {"SIM", "PRED"} is a boolean set to "SIM" indicating that the simulator is used to determine the fitness value (line 6). The value "PRED" is set to indicate that a machine learning model is used as a surrogate to estimate the fitness value. The "PRED" value for variable type is relevant to surrogate-assisted test generation which is discussed in the next section. The dataset DS is populated with the f,l and type for each test input p as shown in the example Figure 3.1 (line 7). Machine learning models usually perform poorly when the dataset is imbalanced as the minority class is not well represented. Moreover, metrics such as Accuracy will be misleading in case of imbalanced datasets. In the case of a heavily imbalanced dataset, the machine learning model can achieve high accuracy by predicting one label for all the test inputs. Therefore, to deal with data imbalance, the well-known SMOTE is applied (line 9). The sampling strategy of SMOTE is set to "minority" which generates as many synthetic minority data points as the difference between the majority and minority class before oversampling. Specifically, for a given dataset with size d such that the number of majority classes is d1 and the number of minority classes is d2 and $d1 + d2 = d$. When the sampling strategy for SMOTE is set to "minority", SMOTE generates $(d1 - d2)$ data points to oversample the minority class thereby balancing the dataset.

To ensure that applying SMOTE to the dataset does not alter the data distribution of the dataset, we discard the labels from SMOTE and instead execute the test inputs using S from SMOTE to compute their actual fitness values and then label the test input as pass or fail based on the actual fitness value. Finally, the remaining execution budget upto 50% execution budget i.e., (Budget/2) is used to extend the dataset iteratively by generating a test input p (line 11) and executing p using S to obtain fitness value f (line

12). The fitness value f is labeled as l (line 13). The variable type is set to "SIM" since the simulator is executed to determine f. Finally, a tuple (f, l, type, p) is added to the dataset DS which is further used in the Main Loop to train machine learning models as described in the next section.

3.3 Main Loop

Recall that we explore two alternative ML-assisted test generation i.e., surrogate-assisted and ML-guided test generation. These two algorithms differ from each other in the main loop with the preprocessing phase remaining the same. The main loop for both algorithms is discussed in the sections below.

3.3.1 Surrogate-Assisted Test Generation

Algorithm 3.2 described surrogate-assisted test generation in which the machine learning models are used to build surrogates that act as a replacement to the actual simulator under test. These surrogates approximate (i.e., predict) the fitness value for a given test input which opens the scope for developing frameworks leveraging surrogates to reduce complete reliance on computationally expensive simulators. The strategy of applying cheap surrogate models to estimate fitness value as an alternative to computationally expensive simulators is adopted in the surrogate-assisted approach where the fitness values of some test inputs are predicted using surrogates. In this way, we do not execute the system for all test inputs. Moreover, the surrogates enable us to explore a larger portion of the input space

Algorithm 3.2 Surrogate-assisted test generation

Input S: The Simulation System
Input R = $\{R_1, \ldots, R_n\}$: Range for input variables v_1 to v_n
Input F: Fitness Function
Input Budget: Maximum number of fitness evaluations
Output DS: A dataset to train failure models

```
1: function DS=SAGEN(S, R, F, Budget)
2:     DS=Preprocess(S, R, F, Budget);                    ▷ Preprocessing Phase
3:     flag = False;
4:     M, del=TRAINSURROGATE(DS);                         ▷ Train surrogate model
5:     while (Budget/2) do
6:     If flag == True
7:     M, del=TRAINSURROGATE(DS);                         ▷ Train surrogate model
8:     end if
9:     p=GEN(R);                                          ▷ Generate random test input
10:    f̂=Predict(M,p);                                    ▷ Predict fitness score for input p
11: If (f(p) ▷ del ̂== PASS&f(p) == P̂ASS)||(f(p) ▷ del ̂== FAIL&f(p) == ̂
    FAIL))
12:    f = f̂;                                             ▷ Prediction is reliable
13:    type = "PRED";
14:    Else
15:    f=SIM(S,p);                                        ▷ Simulate a test input
16:    type = "SIM";
17:    flag = True;
18:    end if
19:    l = Label(f, F);                                   ▷ Label a test input as Pass or Fail
20:    DS = DS ▷ {f, l, type, p};
21:    end while
22:    return DS
23: end function
```

thereby generating larger test sets. This approach can be considered as a framework that is explorative in nature.

Table 3.2: Surrogate models and their descriptions.

Name	Description
GL	Gaussian Process Regression with linear kernel
RT	Regression tree
GNL	Gaussian Process Regression with nonlinear kernel
RF	Random Forest
LSB	Tree-based Gradient Boosting
SVR	Support Vector Regression
NN	A two-layer feedforward Neural Network

Algorithm 3.2 uses dataset DS from the preprocessing phase (Algorithm 3.1) to train a surrogate model SM and evaluate the ML model by computing its error del using Mean Absolute Error (MAE)(line 4). We train the surrogate model SM using 80% of the DS and compute the del using MAE metric on the remaining 20% of DS. The split ratio to divide a dataset for training and testing of ML models is based on the 80/20 rule [47]. The framework then randomly generates a test input p (line 9) and then uses SM to predict its fitness value $\hat{f}(p)$ (line 10). We then determine if the prediction is reliable or not by shifting $\hat{f}(p)$ by the MAE error del (line 11). If the test verdict changes from pass to fail or vice versa after shifting $\hat{f}(p)$, we consider the prediction to be not reliable. If the test verdict does not change after shifting $\hat{f}(p)$ with del, the prediction is considered reliable. The boolean variable "type" is set as either "SIM" or "PRED" depending upon whether the prediction is reliable or not. If the prediction is reliable, we consider $\hat{f}(p)$ to be accurate enough to determine the label of p in the final dataset (line 12), and we set the type to "PRED" indicating that the surrogate model is used to determine the fitness value(line 13). In this case, we do not execute S for p. The test input p along with its predicted

fitness value is added to the dataset. On the other hand, if the prediction is not reliable, we set the type to "SIM" (line 15) which indicates that S needs to be executed to compute the actual fitness value of p. In this case, we add p and the actual fitness value obtained by executing S to the dataset DS (line 16). For every instance where the prediction is not reliable and S is used for execution, we re-train the surrogate model SM using the dataset that includes the new test input and its actual fitness value (lines 6 to 8). It is to be noted that, for each test input, the fitness value is labeled as pass or fail based on the fitness function F. The label (i.e., test verdict) is also added to the dataset for each test input in the dataset DS. The algorithm returns the final dataset DS until the time budget runs out.

For our experimentation, we considered seven surrogate model types as shown in Table 3.2. These machine learning models are the most widely used ones as surrogate models in the evolutionary search and software testing literature [20, 21, 29, 54]. According to the literature and also as we present in our experiment results, no surrogate model from Table 3.2 consistently outperformed other surrogate models for all the requirements subjects under test. Therefore, we propose a novel variation of Algorithm 3.2 in which multiple surrogate models are trained on dataset DS, and the surrogate model with the lowest MAE error is used for prediction. We call it the dynamic surrogate-assisted test generation. The surrogate-assisted and dynamic surrogate-assisted test generation differs in the training phase of the surrogate model (line 7 of Algorithm 3.2). The difference between both the variations is illustrated in Figure 3.2 as a flowchart. More specifically, instead of fitting one ML model that is fixed on the training dataset, we fit seven different surrogate models mentioned in Table 3.2. All seven ML models are then evaluated on the test data. We

then dynamically select the best-performing surrogate model to predict $\hat{f}(p)$ by selecting the ML model with the lowest error. The intuition behind this algorithm is that, as the dataset increases, the performance of a ML model changes. The performance may either increase or decrease. Therefore, rather than restricting the surrogate model to one ML model, dynamically choosing a best-performing ML model from a pool of ML models would lead to accurate predictions. This variation in Algorithm 3.2 allows the algorithm to dynamically select the best surrogate model for each prediction. Every time the prediction is not reliable and the simulator is executed, all the surrogate models are re-trained using the dataset DS with the new test input.

The hyperparameters of all the surrogate models used in the surrogate-assisted test generation are tuned using Bayesian optimization [53]. Hyperparameter tuning is performed while training the surrogate models for the first time after the preprocessing phase. The tuned hyperparameters are fixed for 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 a compute-intensive system. On the other hand, the time taken to re-train a surrogate model (i.e., fit the surrogate model on the train data) without tuning is negligible (i.e., in the order of milliseconds). This prevents the overall approach from becoming prohibitively time-consuming. The overhead of re-training surrogate models does not reduce the performance compared to other alternatives.

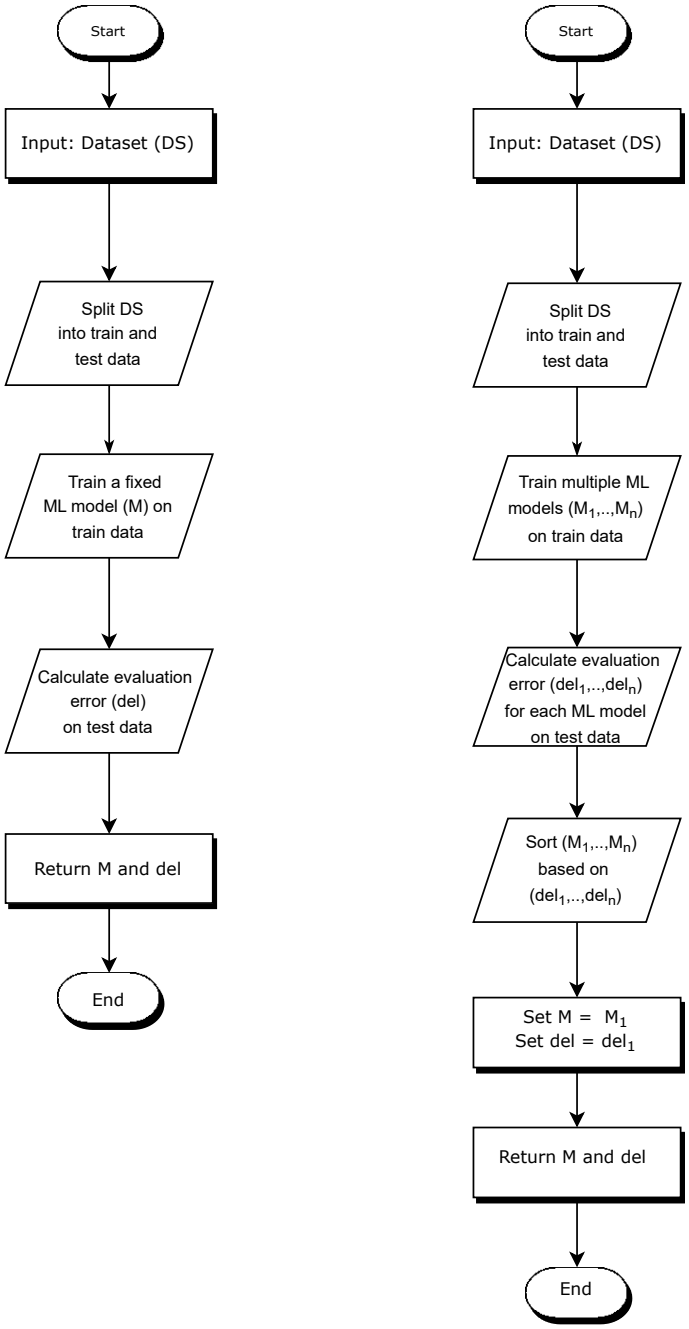

Figure 3.2: Comparison of surrogate-assisted and dynamic surrogate-assisted test generation for training and selecting surrogate model for prediction.

3.3.2 ML-Guided Test Generation

Unlike surrogate-assisted test generation, ML-guided test generation does not perform prediction on fitness value. Instead, ML-guided test generation applies ML on the test dataset to infer boundary regions between pass test inputs and fail test inputs. The purpose of identifying the boundary is to iteratively guide the sampling such that more test inputs are generated in the boundary regions. The idea behind shifting the sampling towards the boundary separating the pass test inputs and fail test inputs is to train ML models on more informative input search space regions by focusing the sampling on regions where neither fail nor pass would be dominant. There are two main benefits in this approach: (1) The ML models learn better to differentiate between pass and fail test inputs. This is because generating more test inputs near the boundary that separates pass and fail test inputs enables building failure models that characterize the test inputs near the boundary. The rules representing pass and fail test inputs on the boundary can be realized to differentiate between pass and fail test inputs.

(2) Generating test inputs near the boundary can be useful to understand test inputs that are moderately failing but a small change in the test input could potentially result in the modified test input passing the requirement. This comparative analysis between the original fail test input and slightly modified pass test input allows the CPS engineers to potentially identify the root cause of the failure in the original test input. To extract useful information from the ML models, the ML models must be interpretable. To help us guide sampling towards the boundary regions, we investigate two alternative ML models: logistic regression (Algorithm 3.3) and regression tree (Algorithm 3.4). A logistic regression model

infers a linear formula over input variables separating the pass test input and fail test inputs while a regression tree approximates pass-fail boundaries in terms of predicates/constraints over input variables. Unlike surrogate-assisted test generation, ML-guided test generation is not used to generate more test data for a given execution budget, but to perform eficient sampling of test inputs for a given execution budget. This approach can be considered as exploitative in nature as it involves sampling test inputs over a specific region of interest in the input search space.

3.3.2.1 ML-guided test generation using Logistic Regression

Algorithm 3.3 describing the logistic regression approach of ML guided test generation starts by taking dataset DS from the preprocessing phase as input to train a logistic regression model (line 4). Since logistic regression is a classification algorithm, the target feature is the label of the test input and the input features are the test input p. The trained logistic model is a linear plane defined by the equation:

$$\log\left(\frac{p}{1-p}\right) = c + \sum_{i=1}^{n} c_i v_i \tag{3.1}$$

In Equation 3.1, $\{v_1, \ldots, v_n\}$ are the input variables, c_i's and c are coeficients, and p is the probability of the pass class [2, 60]. The logistic regression plane is trained on dataset DS and the regression equation is constructed by extracting the coeficients for each input feature from the trained logistic regression model. The value of p is set as the percentage of pass labels in dataset DS and can range between 0 and 1. This formula approximates the region that includes a combination of pass and fails test inputs. As the value of p is

Algorithm 3.3 ML-guided test generation (Logistic regression)

Input S: The Simulation System
Input R = $\{R_1, \ldots, R_n\}$: Range for input variables v_1 to v_n
Input F: Fitness Function
Input Budget: Maximum number of fitness evaluations
Input k: Number of test inputs randomly generated to calculate their Euclidean distance from the logistic regression plane
Output DS: A dataset to train failure models

```
1: function DS=LogRegGEN(S, R, F, Budget, k)
2:     DS=Preprocess(S,R,F,Budget);                    ▷ Preprocessing Phase
3:     while (Budget/2) do
4:     logreg = TRAINLOGREG(DS);                       ▷ Train logistic regression model
5:     P = [];
6:     for i ← 1 to k do
7:     p_i = GEN(R)                                    ▷ Generate k test inputs randomly
8:     P_i = p_i;
9:     end for
10:    t = GENCLOSETOBOUNDARY(logreg,P_i)
11:    f=SIM(S,t);                                     ▷ Simulate a data point
12:    l = Label(f, F);                                ▷ Label test input as pass and fail
13:    type = "SIM";
14:    DS = DS ▷ {f, l, t, type};
15:    end while
16:    return DS
17: end function
```

increased, the boundary plane is set such that more pass test inputs are found on both sides of the plane. That is, the boundary plane is moved towards the region where pass inputs are present homogeneously. On the other hand, as the value of p is decreased, the boundary plane is set such that more fail test inputs are found on both sides of the plane. That is, the boundary plane is moved toward the region where fail inputs are present homogeneously. Therefore, setting p to the percentage of pass test inputs in DS is a heuristic to identify a region that includes almost an equal mix of pass and fail test inputs. The algorithm then samples k test inputs randomly from the input search space (lines 5 to 9). The Euclidean distance d between the logistic regression plane and each of the k test inputs is calculated as follows. The equation of the logistic regression plane equated to 0 as shown below:

$$(c - \log\left(\frac{p}{1-p}\right)) + \sum_{i=1}^{n} c_i v_i = 0 \tag{3.2}$$

We substitute $\{v_1, \ldots, v_n\}$, i.e., the input features from test input to logistic regression in Equation 3.3 as shown below to calculate the Euclidean distance d between the regression plane and test input data point represented by vector $\{v_1, \ldots, v_n\}$.

$$d = \frac{\left|\sum_{i=1}^{n} c_i v_i\right|}{\sqrt{\sum_{i=1}^{n} c_i}} \tag{3.3}$$

The k test inputs are sorted based on d and the closest test input t to the regression plane is selected (line 10). The test input t is executed by S to calculate the fitness value (line 11) and added to dataset DS along with its fitness value f, test verdict l, and the boolean variable type set to "SIM"(line 12 to 14). The logistic regression model is re-

trained every time a new test input is added to DS(line 4). The algorithm is executed to extend DS until the time budget runs out.

3.3.2.2 ML-guided test generation using Regression Tree

Algorithm 3.4 leverages the dataset DS from the preprocessing phase to train a regression tree model (line 4). The main components of the regression tree are the internal nodes, leaves, and edges connecting different nodes and leaves of the regression tree. The edges of the regression tree model represent predicates $v_i \bowtie c$ such that v_i is an input feature to the regression model, $c \in R$ is a constant number and $\bowtie \in \{\leq, >\}$. The leaves of the tree divide the dataset DS into multiple partitions based on the target variable i.e., fitness value such that the information gain is maximized [60]. Information gain is defined as the measure of how much information a feature provides about a target class. A regression tree calculates the information gain for all the features and chooses the feature with the maximum information gain to split the regression tree further. This process is iteratively performed till the maximum depth of the tree has been reached. The leaf nodes are labeled with the average fitness measure of the test inputs in that leaf. After training the regression tree model on DS, we extract all the decision paths along with the fitness value in each leaf node (lines 5 and 6). Algorithm 3.4 identifies predicates $P_i : \{v_{i1} \bowtie c_{i1}, \ldots, v_{in} \bowtie c_{in}\}$ where n is the number of decision paths from the root node to the leaf of the regression tree. We sort the paths based on the leaf fitness value. We then find the two paths such that the fitness value on the leaf node of $P_i < 0$ and $P_{i+1} >= 0$ (lines 7 to 9). The predicates of the two paths specify the boundary between the pass and fail, and therefore are called boundary predicates. We then simplify the boundary predicates by taking the

Algorithm 3.4 ML-guided test generation (Regression Tree)

Input S: The Simulation System
Input R = $\{R_1, \ldots, R_n\}$: Range for input variables v_1 to v_n
Input F: Fitness Function
Input Budget: Maximum number of fitness evaluations
Input ϵ: Percentage of margin around the constant c
Output DS: A dataset to train failure models

```
1: function RS=RegTreeGEN(S, R, F, Budget,ϵ)
2:     DS=Preprocess(S, R, F, Budget);                          ▷ Preprocessing Phase
3:     while (Budget/2) do
4:     regtree = TRAINREGTREE(DS);                              ▷ Train regression tree model
5:     {sP1, sP2, ..., sPk} = EXTRACTDECISIONPATHS(regtree)     ▷ Extract all the
       paths of regression tree
6:     {leaf1, leaf2, ..., leafk} = EXTRACTLEAFVAL(regtree)     ▷ Extract the leaf
       values of each decision path
7:     If leafi < 0&&leafi+1 >= 0                               ▷ Identify decision paths near boundary
8:     bPath1 = sPi;
9:     bPath2 = sPi+1;
10:    {R'i1, R'i2, ....R'im} = EXTRACTBOUNDARYRANGE(bPath1, bPath2, ϵ)
11:    for i ← 1 to m
12:    R = (R \ {Rij}) ∪ {R'ij};                                ▷ Replace ranges in R with the ranges from the
       regression tree model
13:    end for
14:    p=GEN(R);                                                ▷ Generate random test input
15:    f=SIM(S,p);                                              ▷ Simulate a data point
16:    l = Label(f, F);                                         ▷ Label boundary and Non Boundary data points
17:    type = "SIM";
18:    DS = DS ∪ {f, l, type, p};
19:    end while
20:    return DS
21: end function
```

union of all the boundary predicates such that each input variable can have at most one upper-bound predicate ($v \leq c$) and at most one lower-bound predicate ($v > c$) (line 10). For each predicate $v_{i_j} \bowtie c$, 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 11 to 12). The variables that do not appear in the boundary predicates retain their range from the previous iteration. In this way, guiding the sampling around the boundary predicates of each input variable allows the algorithm to generate more test inputs that are close to the boundary between the pass test inputs and fail test inputs. The algorithm then generates test input p from the constrained input search (line 14), executes S with p (line 15), and adds it along with the fitness value f and the corresponding label l to DS (line 16 to 18).

Both Algorithm 3.3 and Algorithm 3.4 employ Bayesian Optimization for tuning the hyperparameters of logistic regression and regression tree for the first time after the preprocessing phase, and the same hyperparameter configuration is fixed for all future iterations.

3.3.3 Building Failure Models

In the previous section, we saw how the dataset DS from the preprocessing phase is extended in the main loop using surrogate-assisted test generation and ML-guided test generation. This section discusses the details of using the extended dataset from the main loop for building failure models. Recall that the output of the main loop is a set DS of tuples $< F(t), l, type, t >$ where t is a test input, $F(t)$ is its fitness value, l is the label of the test input and can take either pass or fail as value, and finally type indicating whether the simulator was executed or the surrogate was used to determine the fitness value. Since

we have the test input and its label, the dataset from the main loop can be used to learn the characteristics of the pass and fail test inputs. In other words, we perform supervised classification tasks using two alternative approaches : (1) Decision rule Model implemented using RIPPER [18] and (2) Decision Tree Model [49] to train failure models. Both the decision tree model and the decision rule model can handle multiple boundaries between pass and fail test inputs in the ML feature space as both creates multiple partitions of the dataset. The decision rule model generates a set of rules in the form of IF-CONDITION-THEN-PREDICTION rules where CONDITION is a conjunction of predicates over input features and the PREDICTION is the predicted label i.e., pass or fail. The decision tree model creates different subsets of the dataset by constructing a tree of predicates. The main difference between a regression tree and a decision tree is that, in a regression tree, the leaf nodes are represented by a numerical value i.e., fitness value in our case, and in the case of the decision trees, the leaf nodes are represented by the label of the target feature. To summarize, the target feature used for training regression trees is numerical and the target feature used for training a decision tree is categorical.

Trees-based models such as regression trees and decision trees split the data into multiple subsets based on certain cutoff values in the features. The prediction of a decision tree is based on the label of the majority of the test inputs for a given leaf node. Building failure models is especially helpful in characterizing failure circumstances from datasets generated by surrogate-assisted test generation because failure models take a holistic approach in characterizing failure circumstances of the system. Instead of considering each failure test input independently for debugging and analysis, failure models allow the identification of a range of similar failure test inputs. Building failure models can be used to understand

the hidden patterns in the dataset.

3.4 Summary

In this section, we understood two alternative ML-assisted test generation approaches namely surrogate-assisted test generation and ML-assisted test generation to build eficient failure models. The motivation to build failure models was also established. To briefly summarize, the key to building an eficient failure model is twofold: (1) Generating more test data for a given time budget (Explorative) or (2) Generating test data from more informative regions that help characterize the difference between a pass and fail test inputs precisely (Exploitative). Surrogate-assisted test generation aims to satisfy (1) whereas ML-guided test generation aims to satisfy (2). In the next section, we discuss the experiment design explaining the research questions, study subjects, and evaluation metrics to evaluate the performance of surrogate-assisted and ML-guided test generation.

Chapter 4

Evaluation Strategy

In this chapter, Section 4.1 introduces the research questions addressed in this thesis. Further, in Section 4.2, an elaborate description of the study subjects used for experimentation are introduced. In the subsequent Sections 4.3 – 4.4, the experimental design and evaluation metrics for each research question are established. Finally, Section 4.5 discusses the complete information about the software/libraries and the system requirements to run the algorithms proposed in Chapter 3.

At a high level, the research questions are framed around three major areas: (1) Analysing the performance of different variations of surrogate-assisted test generation algorithms that varies based on the surrogate model implemented to identify the most optimal variant. (2) Evaluating the eficiency of ML-guided test generation approach in guiding the sampling towards the desired areas in the search space and (3) Evaluating the usefulness of test datasets generated by both surrogate-assisted and ML-guided test gen-

eration frameworks by building failure models trained using datasets generated by both surrogate-assisted and ML-guided techniques. In our evaluation, we aim to answer three main research questions as mentioned below:

4.1 Research Questions

RQ1. Which surrogate model is the most successful in generating the most accurate test data using surrogate-assisted test generation algorithms in an eficient manner?

Surrogate-assisted test generation (SA) algorithm as described in Algorithm 3.2 can have multiple variations depending upon the ML model used as the surrogate model. Further, the size of the test data generated and the accuracy of the predicted fitness values for all test inputs are highly dependent on the surrogate model. RQ1 aims to identify the best variant of SA such that the generated test data has an optimal trade-off between the size of the test data (i.e., Eficiency) and the accuracy of the surrogate predictions.

RQ2. Can the ML-guided test generation frameworks guide the sampling of test inputs towards regions with a mix of both pass and fail test inputs?

Recall that the intuition of ML-guided test generation is to generate a dataset that is more informative. We call the dataset informative when the test dataset is represented by more test inputs from regions where there is a mix of both pass test inputs and fail test inputs as opposed to test inputs from homogeneous regions where either pass test inputs or fail test inputs are predominantly present. We identify the regions where both types of test inputs exist by using fitness value as a guiding metric to perform sampling which

is further discussed in Section 4.4. RQ2 aims to evaluate the eficiency of ML-guided test generation algorithms in guiding the sampling towards the boundaries in the search space where the pass and fail test inputs have fitness values close to each other.

RQ3. How does the performance of surrogate-assisted test generation and ML-guided test generation compare with baseline and with one another in building accurate failure models?

RQ3 aims to perform a quantitative analysis of the performance of failure models inferred from the datasets generated by the best performing SA variant from RQ1, ML-guided test generation using Logistic Regression (LR), and ML-guided test generation using Regression Tree (RT) from RQ2 and Random Search (RS) discussed in Section 2.3.3.1. The scope of evaluating failure models in this thesis does not consider qualitative analysis. Validating the rules generated by the failure models with the help of human experts with domain knowledge is also an important aspect of evaluation, which is one of our future works.

4.2 Study Subjects

We use a publicly available benchmark of CPS from Lockheed Martin [16] for our experiments. This benchmark provides a set of CPS by Lockheed as verification-and-validation challenge subjects for researchers and quality-assurance tool vendors to develop different software verification approaches for these benchmark CPS. The benchmark comprises 11 different CPS. Out of this, 6 were not useful for evaluation because their requirements

either never failed for any test input or failed for all the test inputs. In both cases, there is no need to characterize the failure circumstances simply because the CPS under test never fails or fails for all possible test inputs. Therefore, in this thesis, we focus on 5 different CPS as described in Table 4.1. Each CPS is associated with one or more requirement subjects where each subject has a fitness function that maps the output signals of a CPS to a quantitative fitness value using Restricted Signals First-Order Logic (RFOL) [26], a signal-based logic language to specify CPS requirements used to determine whether a given test input passed or failed the requirement. Although our study subjects focus on Simulink, it is important to note that the surrogate-assisted and ML-guided test generation techniques can be applied to any CPS developed using other numerical-computing environments.

Table 4.1: The requirement subject names, names of the associated benchmark Simulink, descriptions of Simulink, and the number of requirements subjects considered in this thesis. The Simulink are also indicated if it is CI.

Subject Name	Simulink Name	Description of Simulink	#Reqs	CI
TU1...TU9	Tustin	Flight control utility model for computing the Tustin Integration with 57 blocks.	9	✗
REG	Regulator	Regulator used for feedback control applications with 308 blocks.	1	✗
NLG	Nonlinear Guidance	Nonlinear algorithm for generating a guidance command for an air vehicle with 373 blocks.	1	✗
FSM	Finite State Machine	Finite state machine to enable autopilot mode in hazardous situations with 303 blocks.	1	✗
AP1, AP2, AP3	Autopilot	Single-engine, propeller-driven aircraft simulation with 1549 blocks.	3	✓

4.3 Experimental Setting

Table 4.2 shows the parameters used to run the different ML-assisted test generation algorithms for the experiments to answer RQ1, RQ2, and RQ3.

RQ1 experiment setting: Algorithm 3.2 is implemented with 7 different surrogate models from Table 3.2 independently. Let us call these variants of SA as SA-XX where

Table 4.2: 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 15 subjects.

Parameter Name	Description	SA	RS	LR	RT
maxSimNum	Maximum number of fitness evaluations for a given run for all Simulink 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
maxIterNum	Maximum number of iterations allowed for an algorithm for all Simulink 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
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%

XX uniquely identifies a version of Algorithm 3.2 based on the selected surrogate model. For example, SA-RF represents Algorithm 3.2 with Random Forest used as the surrogate model. Algorithm 3.2 is also implemented with an ensemble of surrogate models. Let us call this variation SA-DYN. Recall that SA-DYN can DYNamically switch between surrogate models for each prediction and hence the name. In total, 8 variations of Algorithm 3.2 are evaluated on 4 benchmarks CPS with 12 requirement subjects in total which are non-compute-intensive i.e., (Single execution time of simulator < 1 second). In this thesis, a CPS is considered compute-intensive if a single execution time takes at least 0.5 minutes. Conducting RQ1 experiments on compute-intensive CPS such as Autopilot would be computationally expensive. On average, a single execution time of Autopilot takes approximately 0.5 minutes. Therefore, we evaluate our algorithms on non-compute-intensive CPS in RQ1 and then introduce Autopilot for RQ2 and RQ3. For each requirement subject in RQ1, we run the 8 SA-XX algorithms for an equal time budget. The predicted labels on the test dataset generated by SA-XX can be either from the CPS under test or the surrogate model depending upon whether the prediction is reliable. To observe the accuracy of the dataset, all the predicted test inputs are executed using the simulator to

realize the actual label. This is conducted solely for RQ1 experimentation and will not be part of the real-world implementation of the surrogate-assisted test generation. The non-compute-intensive CPS used for RQ1 experimentation along with the equal time budget allocated for all the surrogate-assisted algorithms are described in Table 4.3. The best performing SA variant is considered as SA algorithm in RQ2 and RQ3. To account for randomness, the experiment for RQ1 was repeated for 10 runs.

Table 4.3: Time budget (in minutes) allocated to the 8 surrogate-assisted algorithms (SA-XX) variations for non-CI subjects (12 subjects).

Simulink Name	Requirement Subject	Preprocessing	Main Loop	Total
Tustin	TU1...TU9	27 min	28 min	55 min
Regulator	REG	19 min	21 min	40 min
Nonlinear Guidance	NLG	8 min	9 min	17 min
Finite State Machine	FSM	4 min	4 min	8 min

Table 4.4: Time taken for surrogate-assisted (SA), ML-guided test generation (LR and RT), and random baseline (RS) in terms of the number of simulations for an execution time budget of 50%.

Subject: TU1...TU9			
Algorithm	Preprocessing	Main Loop	Total
SA-DYN	300	0	300
RS	300	0	300
RT	300	0	300
LR	300	0	300
Subject: REG			
Algorithm	Preprocessing	Main Loop	Total
SA-DYN	300	0	300
RS	300	0	300
RT	300	0	300
LR	300	0	300
Subject: AP1...AP3			
Algorithm	Preprocessing	Main Loop	Total
SA-DYN	300	0	300
RS	300	0	300
RT	300	0	300
LR	300	0	300

Table 4.5: Time taken for surrogate-assisted (SA), ML-guided test generation (LR and RT), and random baseline (RS) in terms of the number of simulations for an execution time budget of 60%.

Subject: TU1...TU9			
Algorithm	Preprocessing	Main Loop	Total
SA-DYN	300	60	360
RS	300	71	371
RT	300	70	370
LR	300	64	364
Subject: REG			
Algorithm	Preprocessing	Main Loop	Total
SA-DYN	300	60	360
RS	300	90	390
RT	300	83	383
LR	300	78	378
Subject: AP1...AP3			
Algorithm	Preprocessing	Main Loop	Total
SA-DYN	300	60	360
RS	300	60	360
RT	300	60	360
LR	300	60	360

RQ2 experiment setting: We compare the two ML-assisted test generation algorithms with baseline RS to identify the most eficient algorithm to guide the search into search space where the mix of pass and fail tests is maximum. Recall that Algorithm 3.3 and Algorithm 3.4 are designed such that test inputs with fitness values close to each other around the value of 0 are sampled. The fitness functions for all the requirement subjects used in this thesis are formulated such that a negative fitness value indicates a fail test input and a positive fitness value indicates a pass test input. Therefore, sampling around the regions where the fitness value of test inputs is close to 0 can generate tests from the boundaries separating the pass and fail test inputs. Note that the RQ2 experiment was performed on all the requirement subjects shown in Table 4.1 except FSM since FSM is a CPS in which all the test inputs are of type boolean. The time budget in terms of the

Table 4.6: Time taken for surrogate-assisted (SA), ML-guided test generation (LR and RT), and random baseline (RS) in terms of the number of simulations for an execution time budget of 70%.

Subject: TU1...TU9			
Algorithm	Preprocessing	Main Loop	Total
SA-DYN	300	120	420
RS	300	133	433
RT	300	132	432
LR	300	125	425
Subject: REG			
Algorithm	Preprocessing	Main Loop	Total
SA-DYN	300	420	420
RS	300	153	453
RT	300	157	457
LR	300	140	440
Subject: AP1...AP3			
Algorithm	Preprocessing	Main Loop	Total
SA-DYN	300	120	420
RS	300	120	420
RT	300	120	420
LR	300	120	420

Table 4.7: Time taken for surrogate-assisted (SA), ML-guided test generation (LR and RT), and random baseline (RS) in terms of the number of simulations for an execution time budget of 80%.

Subject: TU1...TU9			
Algorithm	Preprocessing	Main Loop	Total
SA-DYN	300	180	480
RS	300	195	495
RT	300	194	494
LR	300	186	486
Subject: REG			
Algorithm	Preprocessing	Main Loop	Total
SA-DYN	300	180	480
RS	300	218	518
RT	300	211	511
LR	300	203	503
Subject: AP1...AP3			
Algorithm	Preprocessing	Main Loop	Total
SA-DYN	300	180	480
RS	300	180	480
RT	300	180	480
LR	300	180	480

Table 4.8: Time taken for surrogate-assisted (SA), ML-guided test generation (LR and RT), and random baseline (RS) in terms of the number of simulations for an execution time budget of 90%.

Subject: TU1...TU9			
Algorithm	Preprocessing	Main Loop	Total
SA-DYN	300	240	540
RS	300	257	557
RT	300	255	555
LR	300	247	547
Subject: REG			
Algorithm	Preprocessing	Main Loop	Total
SA-DYN	300	240	540
RS	300	283	583
RT	300	275	575
LR	300	266	566
Subject: AP1...AP3			
Algorithm	Preprocessing	Main Loop	Total
SA-DYN	300	240	540
RS	300	240	540
RT	300	240	540
LR	300	240	540

number of simulator executions is set to 600. To account for randomness, the experiment for RQ2 was repeated for 20 runs.

RQ3 experiment setting: We compare the best performing SA algorithm from RQ1, LR and RT algorithms from RQ2. Additionally, we consider a standard random search as a baseline. SA-DYN is compared with 2 guided search algorithms and 1 random search baseline. The 4 algorithms SA, LR, RT, and RS are applied to the 15 requirement subjects from 5 different CPS described in Table 4.1. For each requirement, subject (AP 1 . . . AP 3) of compute-intensive CPS i.e., Autopilot, the 4 algorithms are executed for an equal execution time budget where the time budget is defined in terms of the maximum number of simulator executions allowed for a given run. In the case of non-compute-intensive CPS i.e., Tustin, Regulator, Nonlinear Guidance, and Finite State Machine, the comparison is con-

Table 4.9: Time taken for surrogate-assisted (SA), ML-guided test generation (LR and RT), and random baseline (RS) in terms of several simulations for an execution time budget of 100%.

Subject: TU1...TU9 (Non-compute-intensive subjects)			
Algorithm	Preprocessing	Main Loop	Total
SA-DYN	300	300	600
RS	300	319	619
RT	300	317	617
LR	300	308	608
Subject: REG (Non-compute-intensive subjects)			
Algorithm	Preprocessing	Main Loop	Total
SA-DYN	300	300	600
RS	300	348	648
RT	300	340	640
LR	300	329	629
Subject: NLG (Non-compute-intensive subjects)			
Algorithm	Preprocessing	Main Loop	Total
SA-DYN	300	5	305
RS	300	6	306
RT	300	3	303
LR	300	2	302
Subject: FSM (Non-compute-intensive subjects)			
Algorithm	Preprocessing	Main Loop	Total
SA-DYN	300	5	305
RS	300	54	354
RT	300	51	351
LR	300	38	338
Subject: AP1...AP3 (Compute-intensive subjects)			
Algorithm	Preprocessing	Main Loop	Total
SA-DYN	300	300	600
RS	300	300	600
RT	300	300	600
LR	300	300	600

ducted such that it is valid for compute-intensive models because both surrogate-assisted and ML-guided frameworks are catered to compute-intensive systems. While fixing the time budget equally for algorithms under comparison is reasonable for compute-intensive models under test, for the non-compute-intensive subjects, the same experimental setting may favor RS since the other three algorithms involve an extra overhead of training surrogate models. The overhead time taken to train the surrogate models is negligible when compared to the execution time of compute-intensive subjects. On the other hand, for non-compute-intensive subjects, a large number of test inputs can be generated within the overhead time for training ML models. Therefore, to conduct experiments with non-compute-intensive subjects and still obtain meaningful results that are valid for compute-intensive subjects as well, an approach proposed by Menghi et. al. [43] is followed that uses the execution time of compute-intensive subjects to restrict the number of test inputs each algorithm can execute for non-compute-intensive subjects. Specifically, the algorithm with lower overhead is allowed to generate additional test inputs. Therefore, these algorithms are allowed to generate y additional test inputs such that $y \cdot time_{CI} = t_d$ where $time_{CI}$ is the execution time of compute-intensive subjects and t_d is the difference in the algorithms' overhead time. Note that, the maximum number of additional tests is limited by the single execution time of a compute-intensive simulator as opposed to the single execution time of a non-compute-intensive simulator. In our work, the single execution time of the autopilot simulator is used as a reference to estimate the time budget for different algorithms. In essence, for a given time budget, the maximum number of test inputs that can be executed by SA, RS, LR, and RT is restricted by the number of Autopilot executions that can be performed within the overhead time difference. The detailed time budget allocated for

different compute-intensive and non-compute-intensive CPS is shown in Table 4.9. Further, for those requirement subjects in which the SA algorithm completes the execution time budget, the performance of failure models is studied by varying the execution time budget equally for SA, RS, LR, and RT. Specifically, the execution time budget mentioned in Table 4.9 is considered the maximum time budget allocated for all algorithms. We then vary the execution time budget of the main loop from 50% to 100% for all algorithms to understand the performance comparison among SA, RS, LR, and RT in different execution time budgets. The motivation behind running the RQ3 experiment by varying the time budget is to analyze the performance of the ML model in both surrogate-assisted and ML-guided test generation as the time budget is increased. Ideally, the performance of the ML model should improve as the time budget is increased under the assumption that the ML model is not fit on noisy train data. Specifically, the preprocessing phase always uses 50% of the total execution time budget irrespective of the algorithms while the execution budget for the main loop is changed to vary the total execution time budget. The detailed execution time budget used for experimentation by varying the main loop time budget is described in Table 4.4 – 4.9. The evaluation metrics to answer RQ3 (discussed in Section 4.4) are reported for different execution time budgets. To account for randomness, these experiments for RQ3 were repeated for 20 runs.

4.4 Evaluation Metrics

This section describes the evaluation metrics used to answer the three research questions described in Section 4.1.

RQ1 Evaluation Metric: For RQ1, the accuracy and eficiency of the dataset generated by the different SA-XX algorithms are measured. The correctness of the test inputs' labels is measured to measure the accuracy of the dataset. The eficiency of the test dataset is evaluated by measuring the size of the test dataset including both test inputs executed by the simulator and test inputs in which the fitness values were predicted by the surrogate model.

RQ2 Evaluation Metric: For RQ2, the test inputs with fitness values around 0 are measured. We consider a threshold $t = 10\% \cdot r$ where r is each dataset's fitness values range. We then create a subrange r_{sub} of fitness values where $r_{sub} \in ((0 - t), \ldots, (0 + t))$. The percentage of test inputs with fitness values within the range r_{sub} concerning the total dataset size is calculated as a metric to observe the number of test inputs near the boundary separating pass and fail test inputs.

RQ3 Evaluation Metric: For RQ3, the datasets obtained by SA, LR, RT and RS is applied to train two alternative failure model: Decision Rule Model (DRM) and Decision Tree Model (DTM). The evaluation metrics used to measure both DRM and DTM are Accuracy, Precision for fail class, and Recall for fail class. The definitions of Accuracy, Precision, and Recall are as follows:

Accuracy is defined as the number of correctly predicted test inputs over the total number of test inputs. The equation for Accuracy is as follows:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{4.1}$$

In Equation 4.1, TP refers to True Positives which are predictions in which the model

correctly predicts the pass class. FP refers to False Positives which are predictions in which the model incorrectly predicts the pass class. TN refers to True Negatives which are predictions in which the model correctly predicts the fail class. FN refers to False Negatives which are predictions in which the model incorrectly predicts the fail class.

Since DRMs are primarily used to predict the failure class, we compute precision and recall for the failure class as per the definition below.

Precision is the ratio of fail class predictions that belong to the fail class. The equation for Precision is as follows:

$$Precision_{Fail} = \frac{TP}{TP + FP} \tag{4.2}$$

In Equation 4.2, TP represents the correctly predicted fail test inputs and FP represents the incorrectly predicted fail test inputs.

Recall is the ratio of fail class predictions made out of all the actual fail tests in the dataset. The equation for Recall is as follows:

$$Recall_{Fail} = \frac{TP}{TP + FN} \tag{4.3}$$

Finally, in equation 4.3, TP refers to the correctly predicted fail test inputs, and FN represents the incorrectly predicted pass test inputs.

4.5 Software and System Requirements

The complete implementation and evaluation of algorithms were performed using MATLAB scripts developed on MATLAB R2021b software that interacts with various CPS benchmarks and Python scripts. It is also ensured that all the Simulink files (.slx) are compatible with the MATLAB version (i.e., R2021b). The Python environment connected to the MATLAB scripts uses Python v3.9.13. All the Python packages were installed using pip - the package installer for Python. The Python packages used are imbalanced-learn v0.8.1 for handling data imbalance using SMOTE, and scikit-learn v1.0.1 to implement all the ML models for both regression and classification tasks except Decision Rules. To implement Decision Rules using RIPPER, a Python package called Wittgenstein v0.3.2 was implemented. Further, scikit-optimize v0.8.1 was implemented to perform Bayesian Search Optimization hyperparameter tuning for all the ML models implemented in Python. MATLAB-based ML models were tuned using the Bayesopt function in MATLAB. All the experiments were performed using Dell Latitude 7420 with Intel i7 CORE vPRO processor Central Processing Unit (CPU) and 16 GB Random Access Memory (RAM).

4.6 Summary

In this chapter, we first discussed the research questions addressed in this thesis. We then explored the study subjects used for evaluation and realized the different compute-intensive and non-compute-intensive requirement subjects. After that, we established the experimental setting used to conduct experiments for each research question. Finally, the

evaluation metrics are clearly stated to systematically answer all the research questions. In the next chapter, we report all the experiment results and subsequently answer the research questions based on the results.

Chapter 5

Evaluation Results and Analysis

In this section, we present the evaluation results from different experiments performed to answer the research questions from section 4.4.

5.1 RQ1 Results

5.1.1 Comparing Accuracy and Eficiency of datasets generated by different SA algorithms

Figure 5.1 shows the accuracy of the dataset defined by the number of incorrectly labeled tests (y-axis) against the dataset size (x-axis) for different surrogate-assisted algorithms. Each data point in the scatterplot represents the performance of the surrogate-assisted algorithm on a given requirement. Recall that there are a total of 12 requirements for

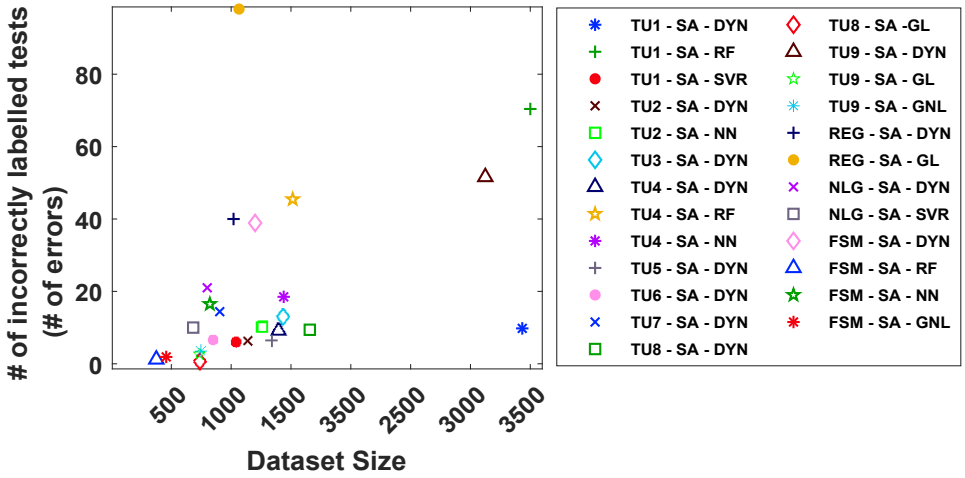

Figure 5.1: Accuracy and eficiency of surrogate-assisted algorithms.

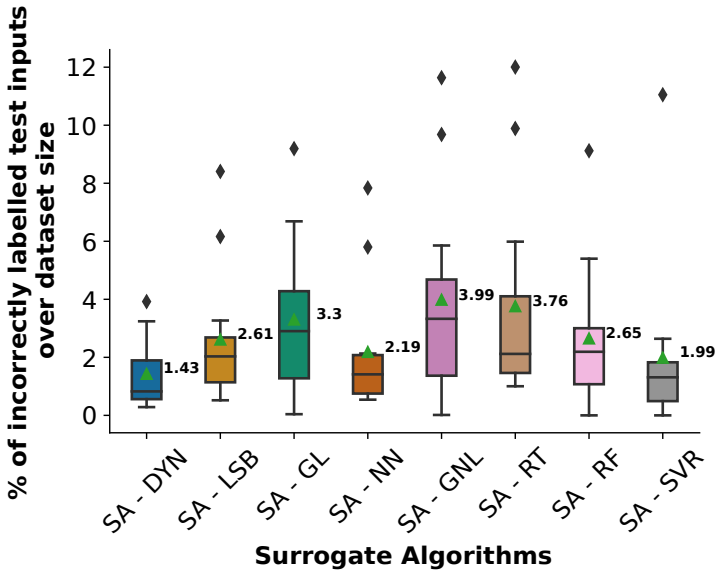

Figure 5.2: Percentages of the incorrect label over the dataset size for different surrogate-assisted algorithms.

non-compute intensive Simulink that are indicated by TU1 to TU9, REG, NLG, and FSM. For each requirement, an ideal surrogate-assisted algorithm is the one with high accuracy (i.e., Lowest value of the y-axis) and high eficiency (i.e., Highest value of the x-axis). Our experiments perform a comparative analysis on a total of 8 surrogate-assisted algorithms with 12 requirements summing up to 96 data points to be represented in the scatterplot. To improve the readability of the scatterplot by reducing clutter, we only present the data points that form the Pareto frontier. More specifically, we only show the points that are not dominated by other competing algorithms concerning either accuracy or dataset size. According to Figure 5.1, SA-DYN is constantly included as the Pareto frontier points for all 12 requirements. Further, SA-DYN is the only best Pareto frontier data point for four subjects, and for the remaining 7 requirements, SA-DYN is one of the multiple best data points that form the Pareto frontier. Overall, SA-DYN offers the best trade-off between the accuracy and eficiency of the dataset. That is, as the size of the dataset increases, the accuracy of the dataset is compromised relatively lower for SA-DYN than the other surrogate-assisted algorithms. For example, in the case of TU1, SA-DYN has better accuracy compared to SA-RF with a difference of 60 label changes while SA-DYN was able to generate almost the same number of data points as SA-RF. (3500 vs 3434 dataset size). Similarly, for the same requirement, SA-DYN has a slightly lower accuracy (10 vs 6 label changes).

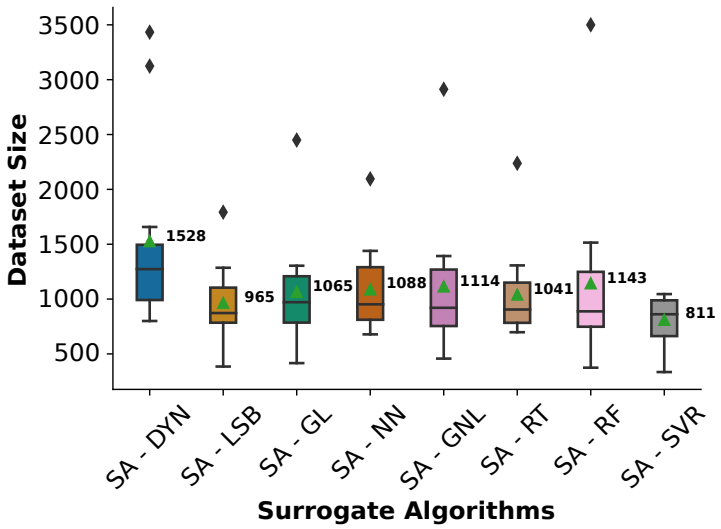

Figure 5.3: Dataset size comparison for different surrogate-assisted algorithms.

5.1.2 Comparing %Error of datasets generated by different SA algorithms

Figure 5.2 presents a boxplot representing the %Error defined as the percentage of incorrect labels over the dataset size for different SA algorithms. The x-axis represents the 8 surrogate algorithms with a boxplot for each SA algorithm. The y-axis defines the %Error which encapsulates the trade-off between the Accuracy of the dataset and the eficiency of the dataset. An ideal SA algorithm aims to have the lowest %Error. As Figure 5.2 depicts, SA-DYN has the lowest Error Percentage when compared to other SA algorithms. This means that SA-DYN can generate more test data with relatively higher accuracy when compared with other SA algorithms. Additionally, the results in 5.2 are tested for statistical significance using the Wilcoxon rank sum test [42] with the level of confidence (α) assigned as 0.05. This means that the hypothesis testing is performed with a confidence

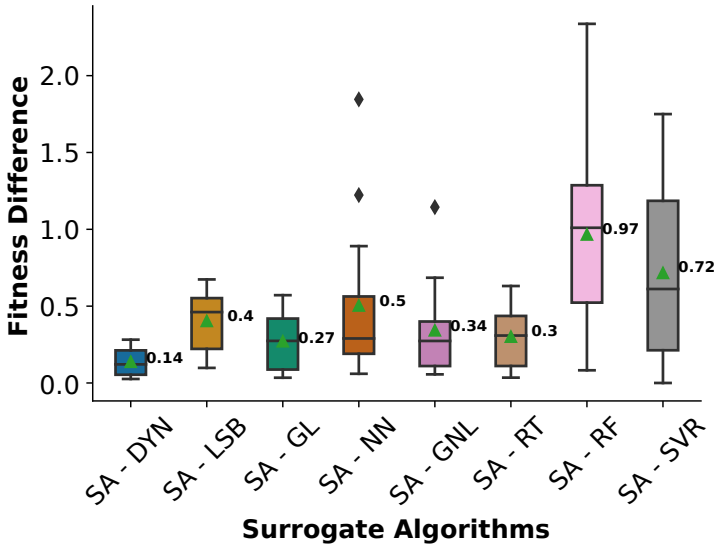

Figure 5.4: Fitness Difference for different surrogate-assisted algorithms.

of 95%. We resort to the same setting for all the hypothesis testing results presented in this work. Based on Wilcoxon rank sum test results, SA-DYN is significantly performing better than the other SA algorithms with a high effect size for SA-GPL, SA-GPNL, and SA-RT; a medium effect size for RT, and a low effect size for SA-NN and SA-RF, and negligible effect size for SA-SVR.

5.1.3 Comparing dataset size of datasets generated by different SA algorithms

Figure 5.3 shows the size of the dataset generated by different SA algorithms for all the 12 requirements. Based on Figure 5.3, we can conclude that the size of the dataset generated by SA-DYN is significantly higher than all the other SA algorithms. More specifically, SA-DYN generates atleast 33% more data in comparison with other SA algorithms. Further,

statistical significance testing using the Wilcoxon rank sum test was performed on the dataset size and it was found that there is a significant difference in the dataset size with a large effect size for SA-SVR, medium effect size for SA-LSB and small effect size for SA-GPL, SA-GPNL, SA-NN, SA-RT and SA-RF.

5.1.4 Comparing Fitness Difference of datasets generated by different SA algorithms

Figure 5.4 represents the Fitness Difference for all the SA algorithms over 12 requirements. Recall that Fitness Difference is defined as the average difference between the predicted fitness value approximated by the surrogate model and the actual fitness value obtained from the simulation. The average is taken over all the predicted test inputs for a given dataset. In 5.4, the x-axis represents the different surrogate models and the y-axis represents the Fitness Difference. Figure 5.4 shows that SA-DYN has the lowest fitness difference in comparison with other SA algorithms. More specifically, there is atleast 50% improvement from other SA algorithms. Here, the word "at least" is used to denote that the performance improvement of SA-DYN is 50% more when compared to the second-best performing SA variant i.e., SA-GL. This indicates that the predictions of fitness score by SA-DYN are more accurate than those obtained from other SA algorithms. Hypothesis testing to compare the fitness difference of different SA algorithms suggests that SA-DYN is significantly better than all the other SA algorithms with a large effect size for SA-RF and SA-SVR, a medium effect size for SA-LSB and SA-RT and small effect size for SA-GPL, SA-GPNL, and SA-NN.

The answer to RQ1 is that the dynamic surrogate-assisted algorithm is significantly more accurate and eficient in terms of generating larger datasets when compared to the surrogate-assisted algorithms that use predefined surrogate models.

5.2 RQ2 Results

5.2.1 Comparing the number of boundary test inputs generated by LR, RT, and RS

Table 5.1 presents the average number of test inputs with a fitness value within a 10% margin of the total fitness range around 0. Let us call the test inputs with fitness values close to 0 (i.e., Within 10% margin) as boundary test inputs. Out of the 14 subjects in total, on average, LR generates more boundary test inputs than RS for 11 subjects except for TU6, AP3, and REG. Likewise, RT generates more boundary test inputs than RS for 12 subjects except for TU6 and REG. Overall, on average over all the requirements, LR generates 266 boundary test inputs, followed by RT which generates 260 boundary test inputs. RS has the least number of boundary test inputs with a value of 227 boundary test inputs. Based on Wilcoxon rank sum test results, all the comparisons LR vs RS, RT vs RS ad LR vs RT are significantly different from each other with a small effect size in all three comparisons.

Table 5.1: Average number of boundary test inputs generated by ML-guided (LR and RT) and random baseline (RS) for 14 subjects with execution time budget set to 600.

Subject	RS	LR	RT
TU1	118	241	259
TU2	203	267	245
TU3	189	296	241
TU4	191	290	235
TU5	209	274	246
TU6	300	288	284
TU7	297	316	297
TU8	231	245	246
TU9	151	155	155
NLG	7	18	15
REG	321	291	305
AP1	210	287	245
AP2	390	398	378
AP3	362	358	492
Average	227	266	260

The answer to RQ2 is that the two ML-guided test generation algorithms (i.e., LR and RT) are more eficient in guiding the search towards regions with a mix of both pass and fail test inputs than the baseline RS.

5.3 RQ3 Results

5.3.1 Comparing Accuracy of DRM and DTM over varying execution time budget

Figures 5.5(a)-(c) – 5.10(a)-(c) is a set of scatterplots representing the accuracy of DRM and DTM independently trained using dataset generated by RS (x-axis) and accuracy of DRM

and DTM trained independently using dataset generated by SA, LR and RT respectively (y-axis) for all the 15 requirements. This experiment is performed for 6 different time budgets ranging from 50% execution time budget to 100% execution time budget. Recall that 50% of the total execution time budget is allocated for the preprocessing phase. Therefore, varying the execution time budget from 50% to 100% refers to varying the remaining 50% of the execution time budget in the main loop. As an example, 50% execution budget means that SA, LR, RT are compared immediately after the preprocessing phase with baseline RS. A comparison with a 60% execution time budget indicates that a 10% of the maximum execution time budget is set as the time budget for the main loop apart from the 50% execution budget for the preprocessing phase. Likewise, 100% execution time budget indicates that 50% of the execution budget is allocated to preprocessing phase and the remaining 50% is completely allocated to the main loop.

Table 5.2 summarizes the accuracy of DRM and DTM trained using SA, LR, RT and RS for different time budgets. In figures 5.5(a)-(c) – 5.10(a)-(c), marker X represents the accuracy of DTM and marker O represents the accuracy of DRM for a given subject. Since there are 15 requirement subjects in total, each scatterplot consists of 15 markers representing DRM and 15 markers representing DTM. A blue color marker represents subjects for which the accuracy for failure models trained using datasets generated by SA, LR or RT is significantly higher when compared to datasets generated by RS according to Wilcoxon rank-sum test. A red marker indicates a subject with no statistical significance between the accuracy of SA, LR, or RT versus the accuracy of RS.

DRM Accuracy Result: From Table 5.2, we can observe that the average accuracy of SA, RS, LR, and RT increases as the time budget is increased from 50% to 100% with

Execution time budget	Average Accuracy (in %) of DRM				Average Accuracy (in %) of DTM			
	SA	RS	LR	RT	SA	RS	LR	RT
50%	71	65	71	71	84	83	84	84
60%	75	67	72	72	86	86	85	85
70%	77	68	72	73	88	86	86	86
80%	78	70	73	74	89	84	87	87
90%	79	71	74	75	89	85	87	86
100%	83	71	76	78	90	84	86	86

Table 5.2: Average Accuracy of DRM and DTM for all the subjects for varying execution time budget from 50% to 100%

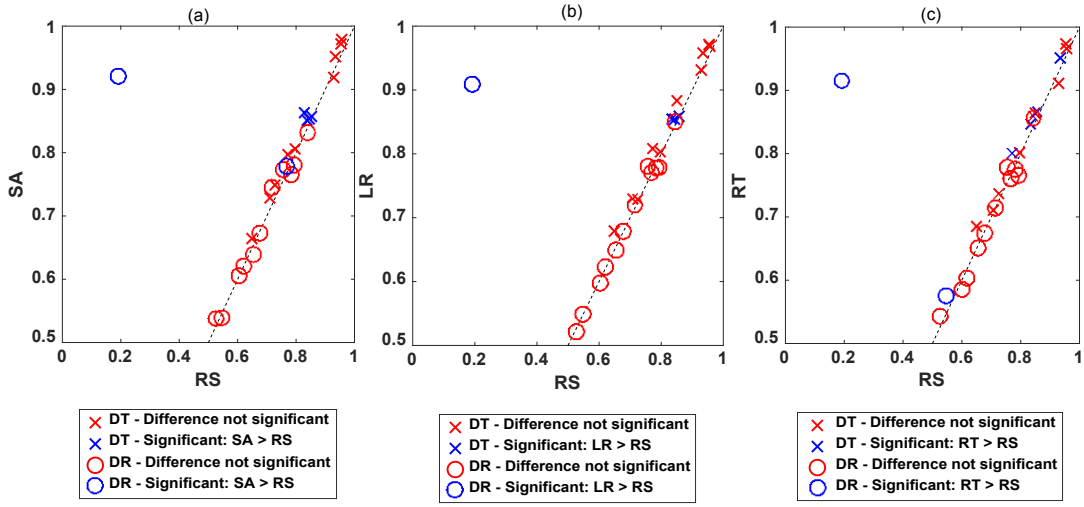

Figure 5.5: Comparing the accuracy of DRM and DTM obtained based on the datasets generated by SA, LR, and RT with those obtained by RS for the 15 subjects in Table 4.1 using 50% execution time budget.

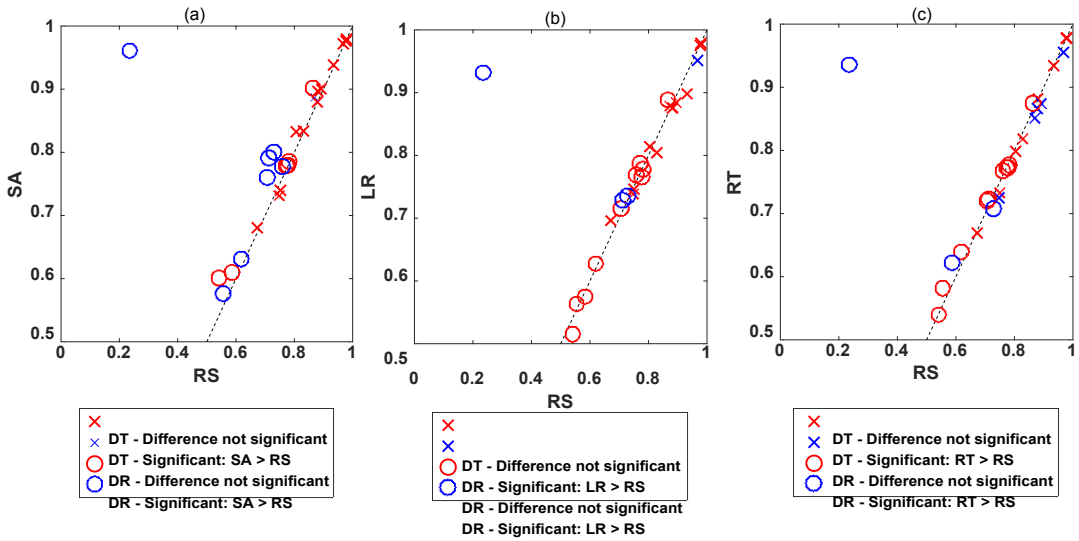

Figure 5.6: Comparing the accuracy of DRM and DTM obtained based on the datasets generated by SA, LR, and RT with those obtained by RS for the 15 subjects in Table 4.1 using 60% execution time budget.

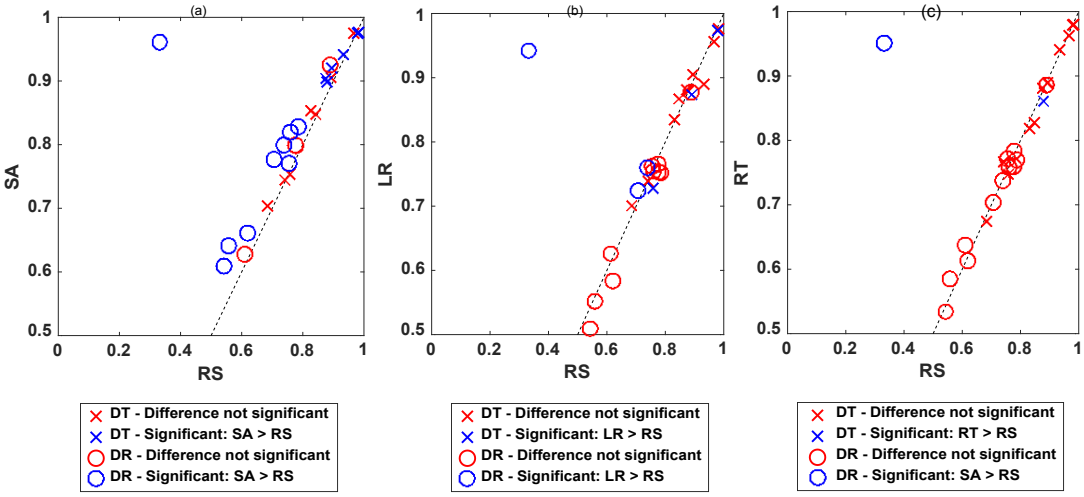

Figure 5.7: Comparing the accuracy of DRM and DTM obtained based on the datasets generated by SA, LR, and RT with those obtained by RS for the 15 subjects in Table 4.1 using 70% execution time budget.

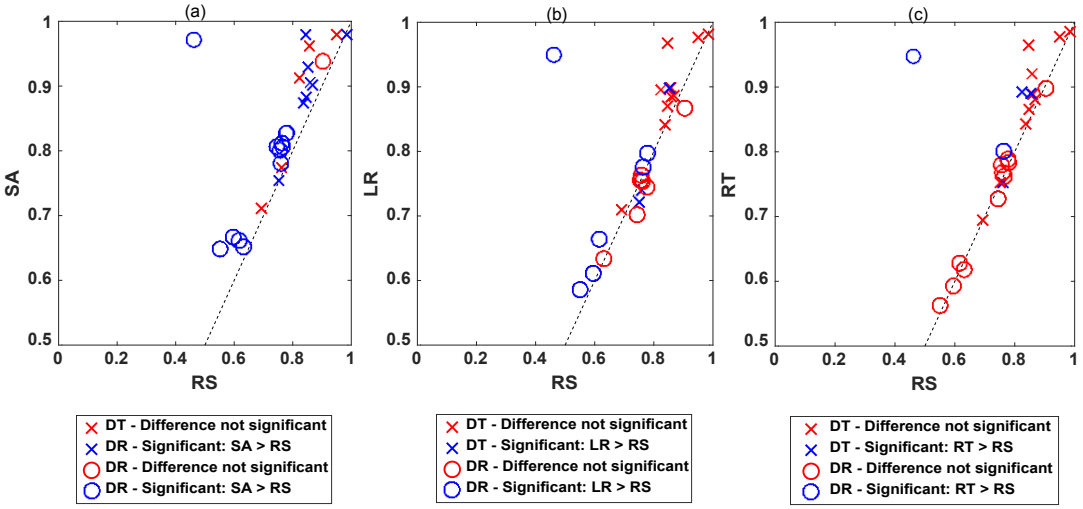

Figure 5.8: Comparing the accuracy of DRM and DTM obtained based on the datasets generated by SA, LR, and RT with those obtained by RS for the 15 subjects in Table 4.1 using 80% execution time budget.

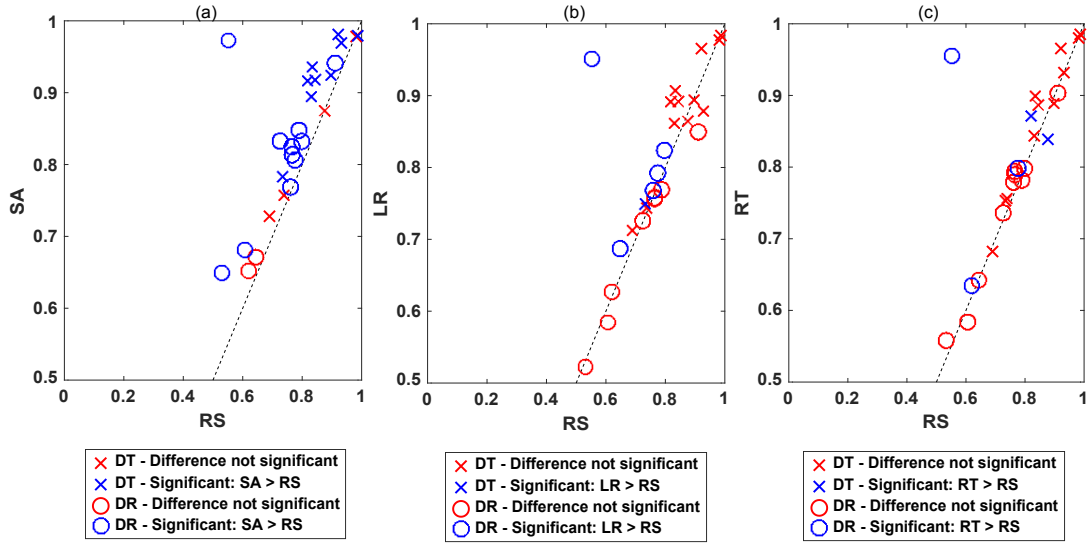

Figure 5.9: Comparing the accuracy of DRM and DTM obtained based on the datasets generated by SA, LR, and RT with those obtained by RS for the 15 subjects in Table 4.1 using 90% execution time budget.

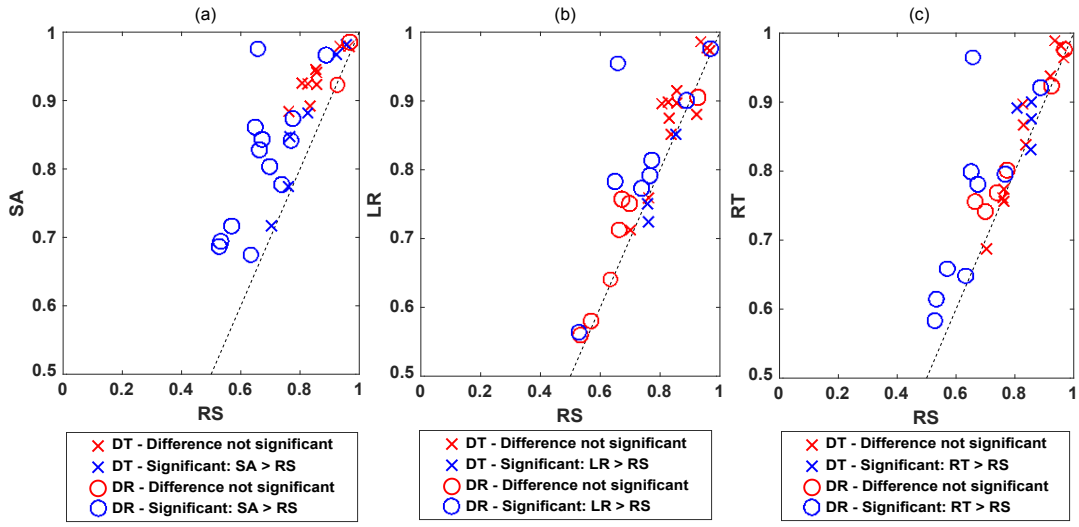

Figure 5.10: Comparing the accuracy of DRM and DTM obtained based on the datasets generated by SA, LR, and RT with those obtained by RS for the 15 subjects in Table 4.1 using 100% execution time budget.

the highest accuracy jump of 12% by SA (71% to 83%), followed by 7% by RT (71% to 78%), 4% by RS (67% to 71%) and finally 4% by LR (72% to 76%). From Figures 5.5(a)-(c) – 5.10(a)-(c), we can note that the number of subjects indicated by blue X markers (i.e., average accuracy of SA, LR or RT significantly higher than average accuracy of RS with DRM as failure model) increases as the time budget is increased from 50% to 100% with the highest jump for SA (7 to 11 subjects), followed by RT (2 to 9 subjects) and finally LR (2 to 8 subjects). Overall, SA has the highest average accuracy (83%), followed by RT (78%) and LR (76%) while RS has the lowest average accuracy (71%) when the algorithms are execution with 100% time budget.

DTM Accuracy Result: According to Table 5.2, the overall trend of the accuracy of SA, RS, LR, and RT increases as the time budget is increased with the highest accuracy

jump of 6% by SA (84% to 90%). The highest accuracy was observed for RS at 70% time budget, LR, and RT at 80% time budget. The average accuracy of RS, LR, and RT drops by 2%, 1%, and 1% respectively as the time budget is increased from 70% to 100%. On average, the accuracy of DTM is greater than the accuracy of DRM for all the time budgets. Figures 5.5(a)-(c) – 5.10(a)-(c) shows that the O markers colored blue in color increases for SA (4 to 9 subjects) as the time budget are increased from 50% to 100% while the number of significant subjects remains the same for LR (3 subjects) and RT (4 subjects). Overall, SA has the highest average accuracy (90%), followed by RT and LR(86%). RS has the lowest average accuracy (84%) when the algorithms are execution with 100% time budget.

5.3.2 Statistical comparison of Accuracy for DRM and DTM

We performed pairwise Wilcoxon rank sum test [42] to statistically compare the average accuracy of SA, RS, LR and RT for both DRM and DTM. The null hypothesis is as follows:

Null Hypothesis: Given two algorithms A and B for comparison, the distribution of average accuracy over all the subjects for A is equal to the distribution of average accuracy over all the subjects for B.

Note that in the definition of the Null Hypothesis, both A and B can be SA, RS, LR, or RT.

Further, Vargha-Delaney's $\hat{A}_{12}$ effect size [56] is used to quantify the difference between the two distributions under comparison. There are 4 values of effect size derived from the $\hat{A}_{12}$ estimate value. They are Negligible, Small, Medium, and Large in the order of increasing difference between the two distributions. To calculate the $\hat{A}_{12}$ estimate and its

Table 5.3: Comparing the accuracy of DRM trained using surrogate-assisted (SA), ML-guided (LR and RT) and random baseline (RS) algorithms with varying execution time budgets using Wilcoxon rank sum test [42] and the Vargha-Delaney's $\hat{A_{12}}$ effect size [56] for all the subjects.

Execution Time Budget: 50%				
Comparison	p-value	null Hypothesis	A_{12} estimate	Effect size
SA vs RS	0.2920	Accepted	-	-
LR vs RS	0.2120	Accepted	-	-
RT vs RS	0.1550	Accepted	-	-
SA vs LR	0.5567	Accepted	-	-
SA vs RT	0.6769	Accepted	-	-
Execution Time Budget: 60%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	9.74E-10	Rejected	0.6168	Small
LR vs RS	0.9300	Accepted	-	-
RT vs RS	0.0154	Rejected	0.5608	Negligible
SA vs LR	6.73E-10	Rejected	0.5716	Negligible
SA vs RT	1.56E-05	Rejected	0.5522	Negligible
Execution Time Budget: 70%				
Comparison	p value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	2.35E-23	Rejected	0.6547	Small
LR vs RS	0.2942	Accepted	-	-
RT vs RS	0.1293	Accepted	-	-
SA vs LR	2.29E-23	Rejected	0.6298	Small
SA vs RT	3.82E-18	Rejected	0.5999	Small
Execution Time Budget: 80%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	3.05E-26	Rejected	0.6725	Medium
LR vs RS	0.0005	Rejected	0.4755	Negligible
RT vs RS	0.0750	Accepted	-	-
SA vs LR	1.30E-29	Rejected	0.6695	Medium
SA vs RT	1.06E-20	Rejected	0.6109	Small
Execution Time Budget: 90%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	1.37E-24	Rejected	0.6389	Small
LR vs RS	0.0626	Accepted	-	-
RT vs RS	0.2019	Accepted	-	-
SA vs LR	1.54E-31	Rejected	0.6630	Small
SA vs RT	1.38E-22	Rejected	0.6210	Small
Execution Time Budget: 100%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	3.04E-43	Rejected	0.7264	Medium
LR vs RS	0.00013	Rejected	0.59383	Small
RT vs RS	2.01E-18	Rejected	0.6383	Small
SA vs LR	1.16E-40	Rejected	0.6640	Small
SA vs RT	1.45E-31	Rejected	0.6171	Small

Table 5.4: Comparing the accuracy of DTM trained using surrogate-assisted (SA), ML-guided (LR and RT) and random baseline (RS) algorithms with varying execution time budgets using Wilcoxon rank sum test [42] and the Vargha-Delaney's $\hat{A_{12}}$ effect size [56] for all the subjects.

Execution Time Budget: 50%				
Comparison	p-value	null Hypothesis	A_{12} estimate	Effect size
SA vs RS	0.002	Rejected	0.46	Negligible
LR vs RS	0.033	Rejected	0.47	Negligible
RT vs RS	0.002	Rejected	0.46	Negligible
SA vs LR	0.55	Accepted	-	-
SA vs RT	0.67	Accepted	-	-
Execution Time Budget: 60%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SAvsRS	0.270	Accepted	-	
LRvsRS	0.010	Rejected	0.4725	Negligible
RTvsRS	2.77E-05	Rejected	0.4635	Negligible
SAvsLR	0.001	Rejected	0.5446	Negligible
SAvsRT	3.38E-05	Rejected	0.5507	Negligible
Execution Time Budget: 70%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	0.001	Rejected	0.5350	Negligible
LR vs RS	0.077	Accepted	-	-Negligible
RT vs RS	0.023	Rejected	0.4801	Negligible
SA vs LR	2.71E-07	Rejected	0.5568	Negligible
SA vs RT	1.04E-06	Rejected	0.5517	
Execution Time Budget: 80%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	2.00E-05	Rejected	0.5376	Negligible
LR vs RS	0.0005	Rejected	0.4619	Negligible
RT vs RS	0.0005	Rejected	0.4642	Negligible
SA vs LR	1.24E-14	Rejected	0.5766	Small
SA vs RT	9.51E-12	Rejected	0.5738	Small
Execution Time Budget: 90%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	3.42E-11	Rejected	0.6769	Small
LR vs RS	0.013	Rejected	0.5354	Negligible
RT vs RS	6.27E-05	Rejected	0.4647	Negligible
SA vs LR	1.24E-15	Rejected	0.5910	Small
SA vs RT	9.92E-18	Rejected	0.5953	Small
Execution Time Budget: 100%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	9.42E-13	Rejected	0.5752	Small
LR vs RS	3.85E-05	Rejected	0.4559	Negligible
RT vs RS	1.93E-06	Rejected	0.4549	Negligible
SA vs LR	2.16E-22	Rejected	0.6064	Small
SA vs RT	1.56E-19	Rejected	0.6022	Small

corresponding effect size, we used an R language-based package called effsize.

Statistical comparison of Accuracy for DRM: Table 5.3 shows the statistical comparison of the average accuracy of DRM trained using SA, RS, LR and RT for different execution time budgets for all subjects. We can observe that for all execution time budgets, the null hypothesis is rejected in all the comparisons where SA is involved (i.e., SA vs RS, SA vs LR, and SA vs RT) except when the execution time budget is set to 50%. We can infer that the accuracy of SA is significantly different from the other three algorithms for all time budgets except 50%. The reason SA, RS, LR, and RT have no statistical significance for the 50% execution time budget is that the datasets from different algorithms are compared immediately after the preprocessing phase. SA, LR, and RT are uniquely identified by different applications of ML models in the main loop whereas the preprocessing phase remains the same for all algorithms. The null hypothesis is accepted when comparing LR vs RS for the execution time budgets 50% to 90%. We can infer that the accuracy of the LR is significantly better than SA only when the time budget is set to maximum. A similar trend is observed when comparing RT and RS in which the null hypothesis is accepted for the time budgets 70%, 80%, and 90%. Note that when the execution time budget is set to 100%, the following observations can be made: The null hypothesis for all the comparisons (i.e., SA vs RS, LR vs RS, RT vs RS, SA vs LR, and SA vs RT) was rejected indicating that the performance of SA, RS, LR and RT in terms of average accuracy were significantly different from each other with an effect size of Medium for SA vs RS and effect size of Small for all other cases.

Statistical comparison of Accuracy for DTM: Table 5.4 presents the Wilcoxon rank sum test results to compare the average accuracy of DTM trained using SA,RS,LR

Execution time budget	Recall (in %) of DRM				Recall (in %) of DTM			
	SA	RS	LR	RT	SA	RS	LR	RT
50%	84	90	84	85	74	75	75	75
60%	86	91	85	86	77	75	75	75
70%	88	90	87	87	79	77	77	77
80%	88	90	88	88	81	75	78	78
90%	88	86	88	88	83	78	79	79
100%	88	83	87	85	83	79	79	79

Table 5.5: Recall of DRM and DTM over all the subjects for varying execution time budget from 50% to 100%

and RT for different time budgets for all subjects. Based on the results, we can understand that the null hypothesis is rejected when comparing SA vs RS (i.e., Accuracy in case of SA is significantly higher than Accuracy of in case of RS) for all time budgets except 60% with effect size varying from Negligible to Small. For the comparison between LR vs RS and RT vs RS, the null hypothesis is rejected for all the time budgets with Negligible effect size. Finally, for the comparison of SA vs LR and SA vs RT, the hypothesis is rejected for all time budgets.

5.3.3 Comparing Recall of DRM and DTM over varying execution time budget

Recall as a metric is used to measure the ability of the DRMs to precisely identify the condition that results in a failure. A high recall indicates that the DRM can cover the maximum number of failure test inputs using the conditions that it learns from a dataset. Table 5.5 presents the recall of DRM and DTM trained using SA, LR, RT, and RS for

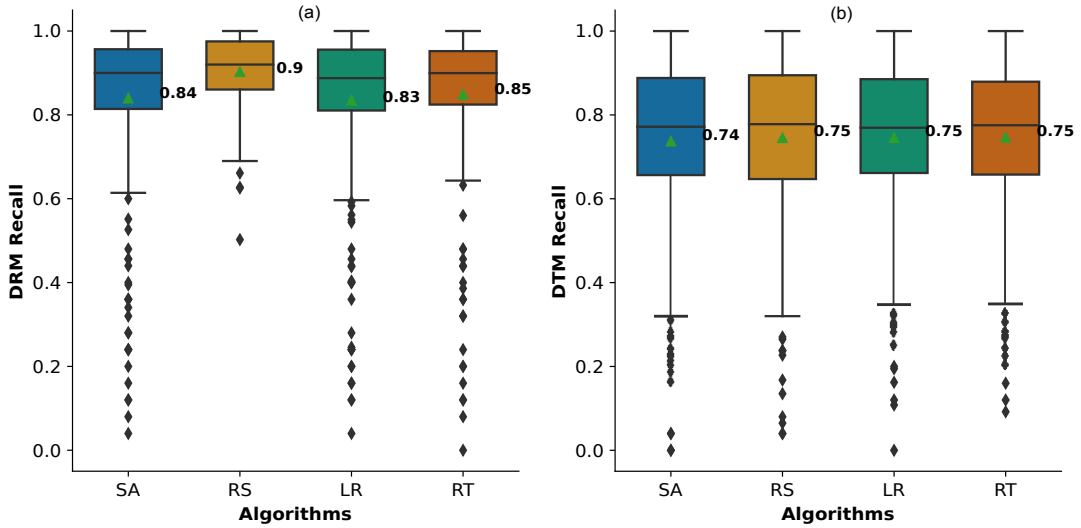

Figure 5.11: Recall for the DRM and DTM obtained from the datasets generated by SA, RL, RT and RS for all 15 subjects using 50% execution time budget.

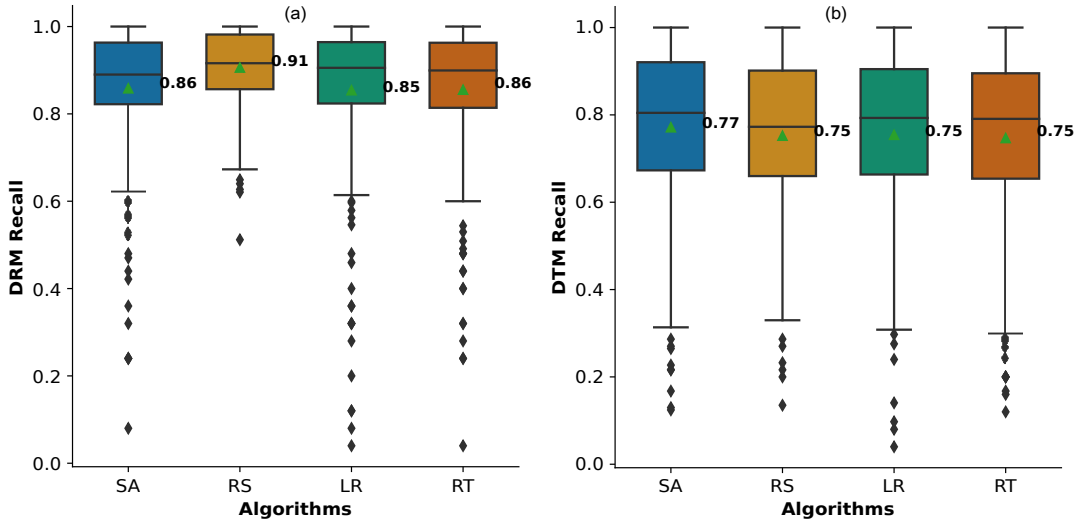

Figure 5.12: Recall for the DRM and DTMobtained from the datasets generated by SA, RL, RT and RS for all 15 subjects using 60% execution time budget.

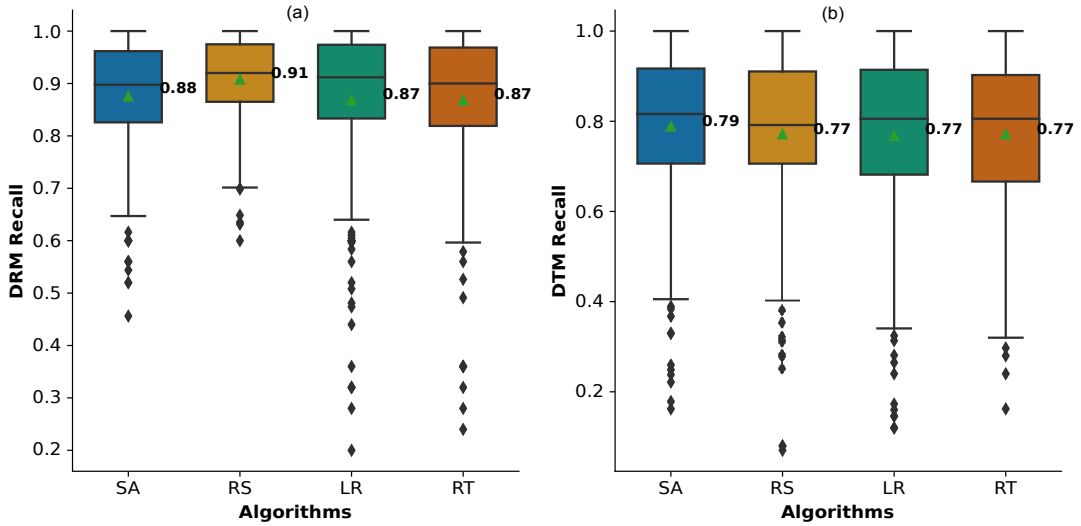

Figure 5.13: Recall for the DRM and DTM obtained from the datasets generated by SA, RL, RT and RS for all 15 subjects using 70% execution time budget.

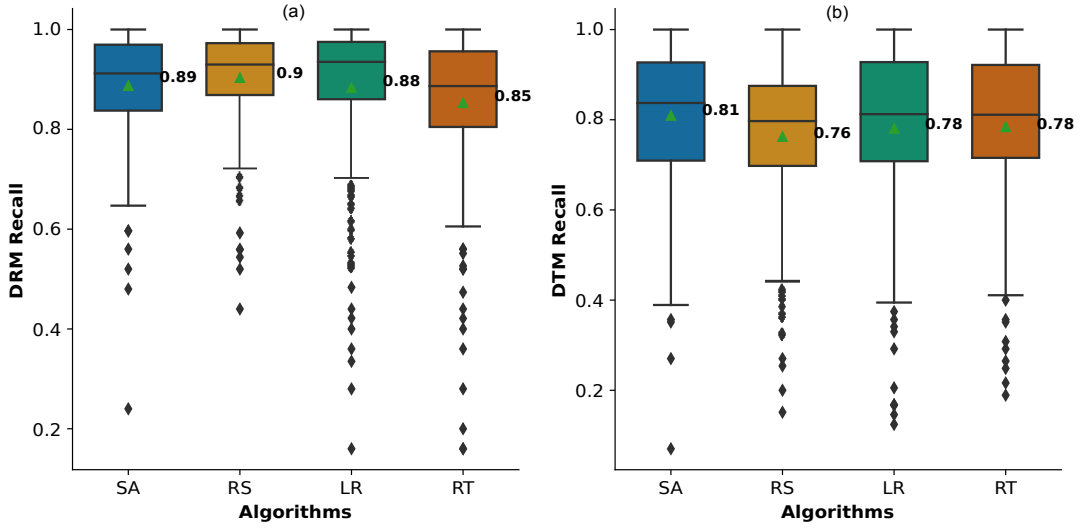

Figure 5.14: Recall for the DRM and DTM obtained from the datasets generated by SA, RL, RT and RS for all 15 subjects using 80% execution time budget.

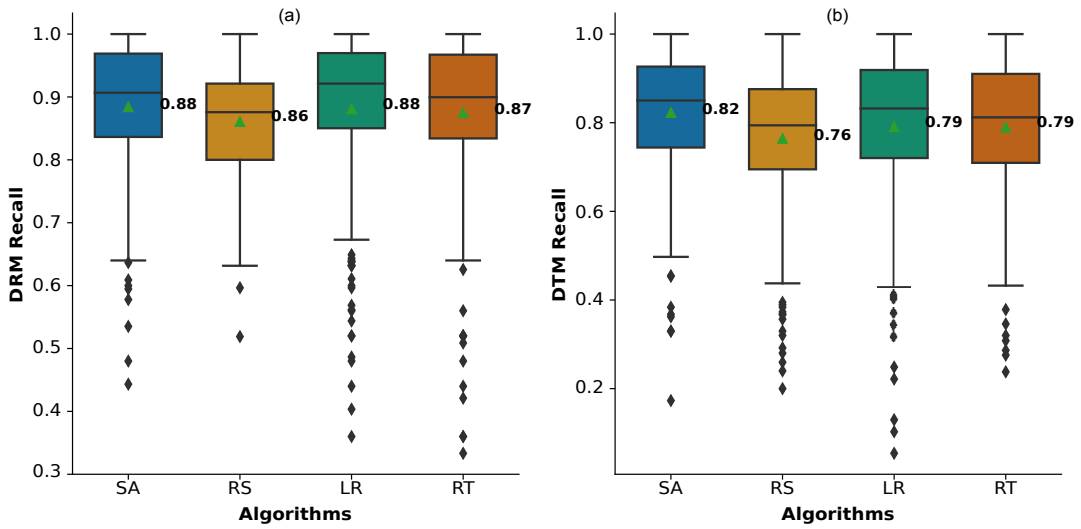

Figure 5.15: Recall for the DRM and DTM obtained from the datasets generated by SA, RL, RT and RS for all 15 subjects using 90% execution time budget.

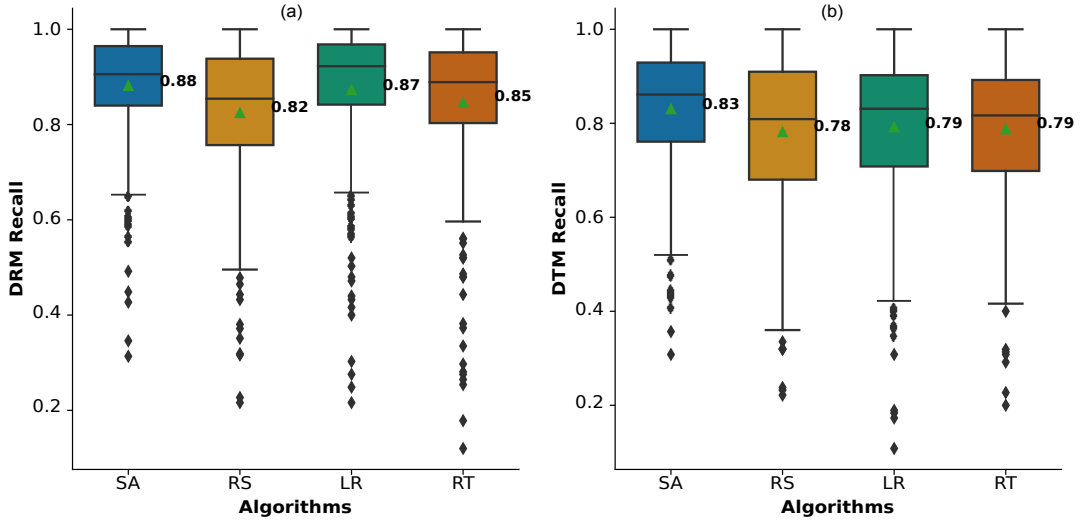

Figure 5.16: Recall for the DRM and DTM obtained from the datasets generated by SA, RL, RT and RS for all 15 subjects using 100% execution time budget.

15 subjects over varying executing time budgets. Figures 5.11 – 5.16 is a set of boxplots representing the recall for DRM and DTM trained using SA, RS, LR, and RT for all the 15 subjects in varying execution time budgets from 50% to 100%.

DRM Recall Result:, we can observe that the recall for SA, LR, and RT increases while the recall for RS decreases as the time budget is increased from 50% to 100%. Specifically, recall increased by 3% for SA (84% to 88%) and LR (84% to 87%) while the recall remained the same for RT (85%). Finally, the recall for RS decreases by 7% (90% to 83%). When the execution time budget is set to 100%, SA has the highest recall (88%) followed by LR (87%) and RT (85%). RS has the lowest recall (83%) when the execution time budget is set to maximum.

DTM Recall Result:, the recall for SA, RS, LR and RT increases as the time budget is increased from 50% to 100%. SA has the highest increase of 9% (74% to 83%) which is followed by 4% (75% to 79%) improvement by LR, RT and RS. When the time budget is set to maximum, SA has the highest recall (83%) while LR, RT, and RS have a recall of 79%. It is interesting to note that the recall of DRM is higher than DTM for all algorithms for all time budgets. Therefore, DRM as a failure model is eficient in uses cases that involve covering as many failure inputs as possible. One of the reasons for this can be attributed to the overlapping rules generated by DRM allowing the decision rules to cover more failure test inputs than the DTM which partitions the dataset without any overlaps.

Table 5.6: Comparing recall of DRM trained using surrogate-assisted (SA), ML-guided (LR and RT) and random baseline (RS) algorithms with varying execution time budget using Wilcoxon rank sum test [42] and the Vargha-Delaney's $\hat{A}_{12}$ effect size [56] for all the subjects.

Execution Time Budget: 50%				
Comparison	p-value	A_{12} estimate	Effect size	
SA vs RS	6.98E-06	Rejected	6.98E-06	Small
LR vs RS	6.52E-09	Rejected	0.3980	Small
RT vs RS	7.87E-05	Rejected	0.4197	Small
SA vs LR	0.4196	Accepted	-	-
SA vs RT	0.260	Accepted	-	-
Execution Time Budget: 60%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	1.38E-06	Rejected	0.4126	Medium
LR vs RS	7.11E-06	Rejected	0.4287	Negligible
RT vs RS	1.66E-07	Rejected	0.4100	Small
SA vs LR	0.4709	Accepted	-	-
SA vs RT	0.8566	Accepted	-	-
Execution Time Budget: 70%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	1.83E-06	Rejected	0.4137	Small
LR vs RS	0.0013	Rejected	0.4523	Negligible
RT vs RS	1.52E-05	Rejected	0.4276	Negligible
SA vs LR	0.9948	Accepted	-	-
SA vs RT	0.9962	Accepted	-	-
Execution Time Budget: 80%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	0.0071	Rejected	0.4475	Negligible
LR vs RS	0.7368	Accepted	-	-
RT vs RS	8.82E-07	Rejected	0.4003	Small
SA vs LR	0.1266	Accepted	-	-
SA vs RT	0.0003	Rejected	0.5508	Negligible
Execution Time Budget: 90%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	0.0002	Rejected	0.4385	Negligible
LR vs RS	0.0057	Rejected	0.4654	Negligible
RT vs RS	3.76E-05	Rejected	0.4332	Negligible
SA vs LR	0.6330	Accepted	-	-
SA vs RT	0.5489	Accepted	-	-
Execution Time Budget: 100%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	1.89E-12	Rejected	0.7319	Medium
LR vs RS	4.64E-11	Rejected	0.6323	Small
RT vs RS	0.003872	Rejected	0.5716	Negligible
SA vs LR	0.679912	Accepted	-	-Negligible
SA vs RT	5.78E-05	Rejected	0.5606	

Table 5.7: Comparing recall of DTM trained using surrogate-assisted (SA), ML-guided (LR and RT) and random baseline (RS) algorithms with varying execution time budget using Wilcoxon rank sum test [42] and the Vargha-Delaney's $\hat{A}_{12}$ effect size [56] for all the subjects.

Execution Time Budget: 50%				
Comparison	p-value	A_{12} estimate	Effect size	
SA vs RS	0.4608	Accepted	-	-
LR vs RS	0.7052	Accepted	-	-
RT vs RS	0.9507	Accepted	-	-
SA vs LR	0.4735	Accepted	-	-
SA vs RT	0.6475	Accepted	-	-
Execution Time Budget: 60%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	0.098	Accepted	-	-
LR vs RS	0.253	Accepted	-	-
RT vs RS	0.026	Rejected	0.4844	Negligible
SA vs LR	0.066	Accepted	-	-
SA vs RT	0.001	Rejected	0.5331	Negligible
Execution Time Budget: 70%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	0.022	Rejected	0.5285	Negligible
LR vs RS	0.680	Accepted	-	-
RT vs RS	0.666	Accepted	-	-
SA vs LR	0.028	Rejected	0.5228	Negligible
SA vs RT	0.004	Rejected	0.5312	Negligible
Execution Time Budget: 80%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	0.001	Rejected	0.5336	Negligible
LR vs RS	0.181	Accepted	-	-
RT vs RS	0.659	Accepted	-	-
SA vs LR	5.66E-06	Rejected	0.5375	Negligible
SA vs RT	1.11E-05	Rejected	0.5453	Negligible
Execution Time Budget: 90%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	4.02E-08	Rejected	0.5590	Negligible
LR vs RS	0.8728	Accepted	-	-
RT vs RS	0.9387	Accepted	-	-
SA vs LR	2.23E-08	Rejected	0.5473	Negligible
SA vs RT	1.05E-08	Rejected	0.5621	Negligible
Execution Time Budget: 100%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	2.35E-08	Rejected	0.6469778	Small
LR vs RS	0.1164	Accepted	-	-
RT vs RS	0.0684	Accepted	-	-
SA vs LR	1.36E-12	Rejected	0.5737	Small
SA vs RT	5.12E-14	Rejected	0.5925	Small

5.3.4 Statistical comparison of Recall for DRM and DTM

A pairwise Wilcoxon rank sum test was performed to determine the statistical significance of recall among SA, RS, LR, and RT over all subjects. The null hypothesis is as follows:

Null Hypothesis: Given two algorithms A and B, the distribution of recall over all the subjects for A is equal to the distribution of recall over all the subjects for B.

Note that both A and B can be SA, RS, LR and RT.

Statistical comparison of Recall for DRM: Table 5.6 depicts the statistical comparison of recall of SA, RS, LR and RT for different time budgets for all subjects. We can observe that the null hypothesis is rejected for all time budgets when comparing SA vs RS with effect sizes ranging from Negligible to Medium and RT vs RS with effect size Negligible. This indicates that the recall of SA is significantly different from the recall of RS. The null hypothesis is rejected for all time budgets except 80% execution time budget when comparing LR vs RS with an effect size ranging from Negligible to Small. The null hypothesis is accepted for all execution time budgets when comparing SA vs LR. Finally, the null hypothesis is accepted for all time budgets except 80% and 100% execution time budgets when comparing SA vs RT.

Statistical comparison of Recall for DTM:Table 5.7 shows the Wilcoxon test results for the comparison of recall of DTM trained using SA, RS, LR and RT for 15 subjects over varying time budgets. We can understand that the null hypothesis is accepted for all pairwise comparisons when the algorithms are executed with 50% time budget indicating that there are no significant differences among the recall of SA, RS, LR and RT. When the algorithms are executed with a 60% time budget, the null hypothesis is rejected for the

Execution time budget	Precision (in %) of DRM				Precision (in %) of DTM			
	SA	RS	LR	RT	SA	RS	LR	RT
50%	54	54	55	55	76	76	75	75
60%	61	54	56	57	78	75	75	75
70%	63	55	57	58	81	79	77	77
80%	65	56	57	59	83	79	79	79
90%	66	56	58	60	85	81	79	78
100%	72	57	62	64	85	81	79	78

Table 5.8: Precision of DRM and DTM over all the subjects for varying execution time budget from 50% to 100%

comparisons SA vs RS and SA vs RT with Negligible effect size. Finally, for the execution time budgets 70% to 100%, the null hypothesis for comparisons SA vs RS, SA vs LR, and SA vs RT are rejected with effect size varying from Negligible to Small.

5.3.5 Comparing Precision of DRM and DTM over varying execution time budget

Precision measures the ability of the DRM to generate failure test inputs correctly. A high precision indicates that the DRM can correctly detect a maximum number of failure test inputs. Table 5.8 shows the precision of DRM and DTM trained using SA, RS, LR and RT for execution time budget varying from 50% to 100% for all the subjects. Figures 5.17 – 5.22 is a set of boxplots representing the precision for DRM and DTM trained using SA, RS, LR, and RT for all the 15 subjects in varying execution time budgets from 50% to 100%. Section 5.3.3 discusses the results comparing the precision of different algorithms respectively.

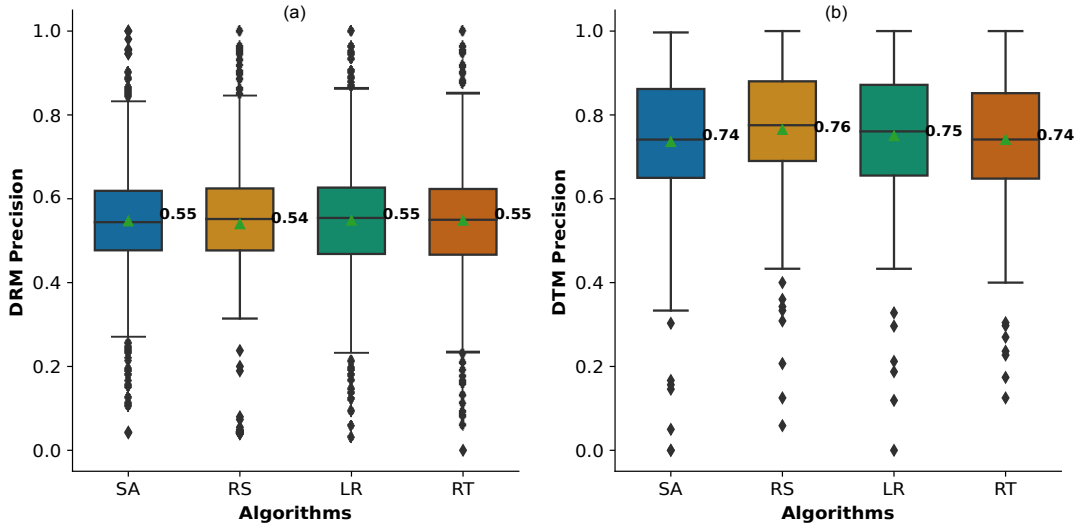

Figure 5.17: Precision for the DRM and DTM obtained from the datasets generated by SA, RL, RT and RS for all 15 subjects using 50% execution time budget.

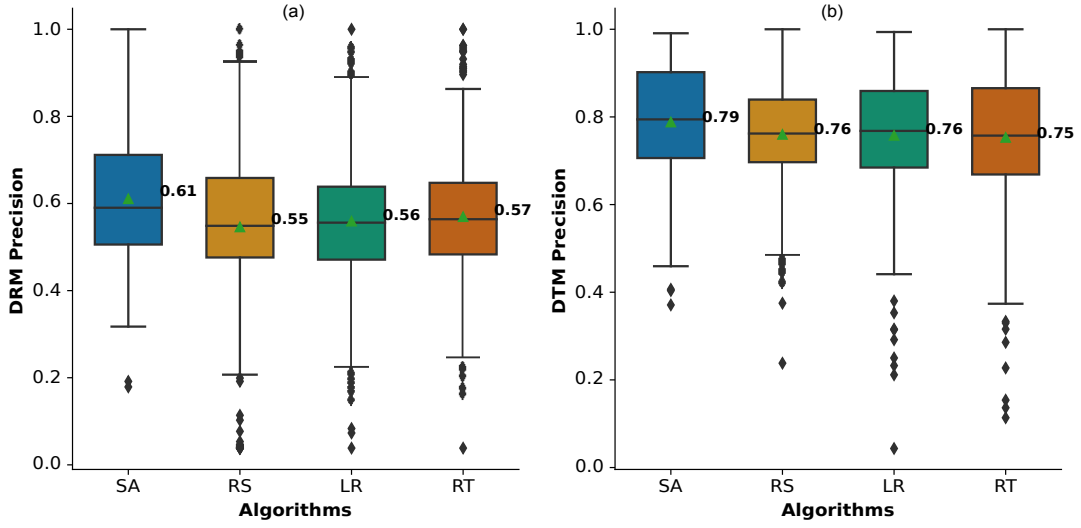

Figure 5.18: Precision for the DRM and DTM obtained from the datasets generated by SA, RL, RT and RS for all 15 subjects using 60% execution time budget.

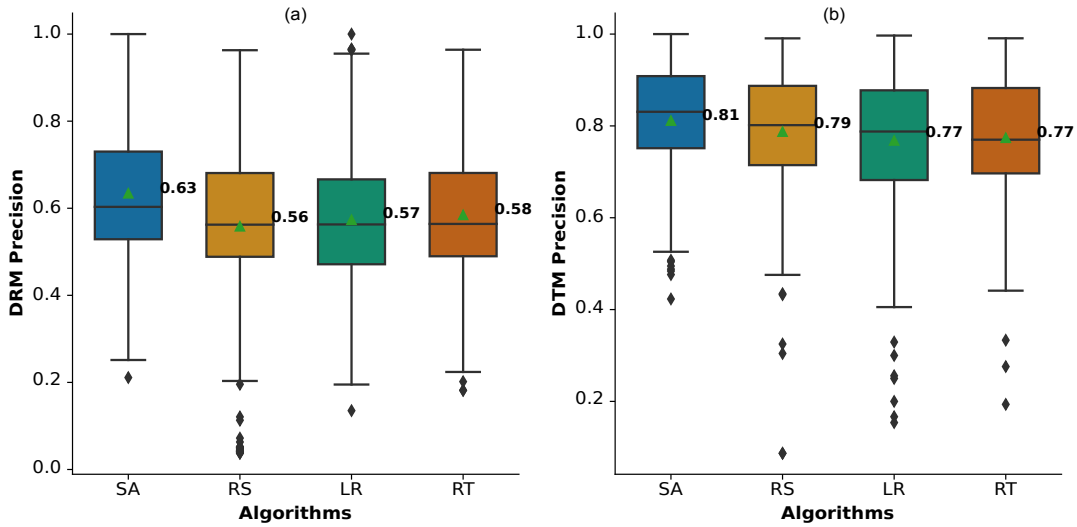

Figure 5.19: Precision for the DRM and DTM obtained from the datasets generated by SA, RL, RT and RS for all 15 subjects using 70% execution time budget.

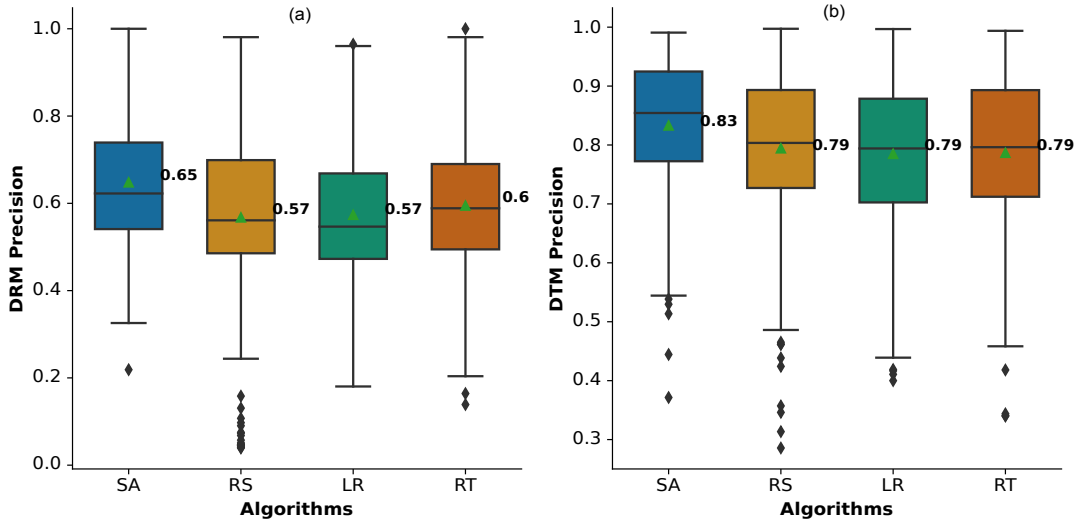

Figure 5.20: Precision for the DRM and DTM obtained from the datasets generated by SA, RL, RT and RS for all 15 subjects using 80% execution time budget.

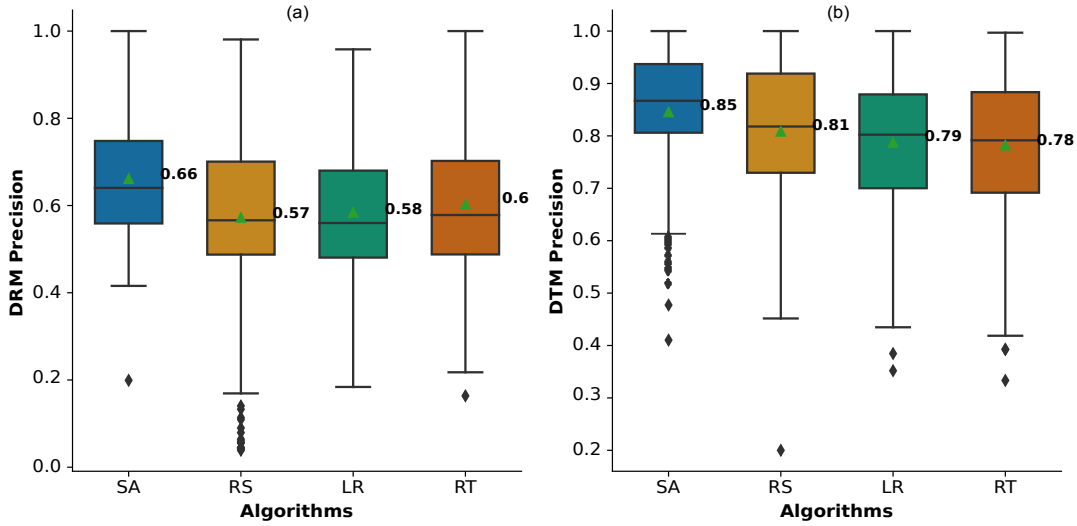

Figure 5.21: Precision for the DRM and DTM obtained from the datasets generated by SA, RL, RT and RS for all 15 subjects using 90% execution time budget.

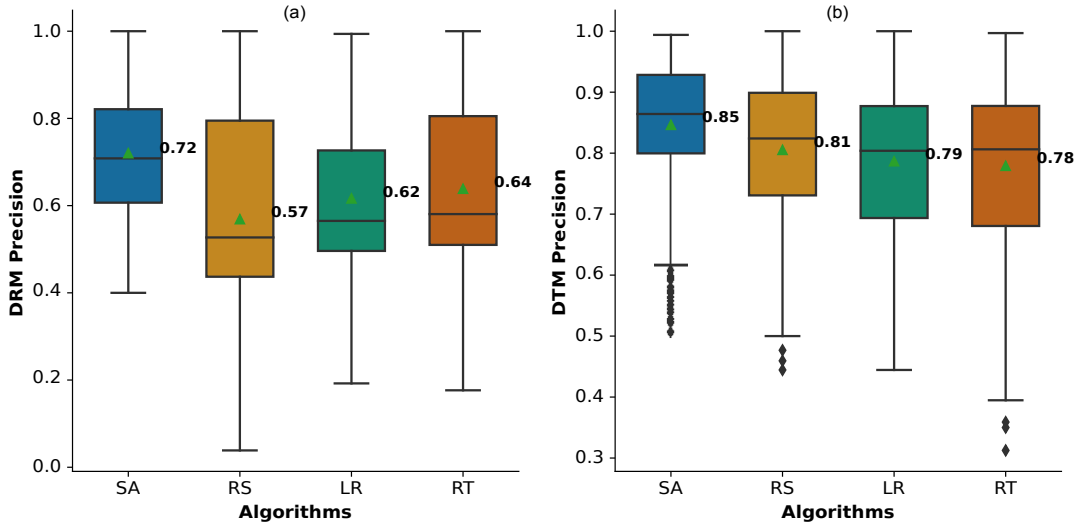

Figure 5.22: Precision for the DRM and DTM obtained from the datasets generated by SA, RL, RT and RS for all 15 subjects using 100% execution time budget.

DRM Precision Result: The observations that can be made: (1) Similar to accuracy, there is an increasing trend for precision as the budget is increased from 50% to 100%. The highest improvement in precision was achieved by SA with a jump of 18% (54% to 72%), followed by RT with a jump of 9% (55% to 64%) and LR with a jump of 7% (55% to 62%) respectively. Finally, RS has the lowest precision improvement of 3% (54% to 57%). (2) For the maximum execution budget, SA (72%) has the highest precision followed by RT (64%), LR (62%) and RS (57%).

DTM Precision Result: The precision for SA, RS, LR and RT increases as the time budget is increased. Specifically, the precision of SA increased by 9% (76% to 85%), SA increased by 5% (76% to 81%), LR increased by 4% (75% to 79%) and RT increased by 3% (75% to 78%). When the budget is set to maximum, SA has the highest precision of 85% followed by RS, LR and RT with a value of 81%, 79%, and 78% respectively. Overall, for all the time budgets, the precision of DTM is higher than the precision of DRM for all algorithms indicating that DTM may be better suited for building failure models to precisely generate new failure tests.

5.3.6 Statistical comparison of Precision for DRM and DTM

Similar to average accuracy and recall, the null hypothesis to perform the Wilcoxon test on the distribution of precision is as follows:

Null Hypothesis: Given two algorithms A and B, the distribution of precision over all the subjects for A is equal to the distribution of precision over all the subjects for B.

Statistical comparison of Precision for DRM: Table 5.9 presents the Wilcoxon

Table 5.9: Comparing precision of DRM trained using surrogate-assisted (SA), ML-guided (LR and RT) and random baseline (RS) algorithms with varying execution time budget using Wilcoxon rank sum test [42] and the Vargha-Delaney's $\hat{A}_{12}$ effect size [56] for all the subjects.

Execution Time Budget: 50%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	0.2722	Accepted	-	-
LR vs RS	0.1655	Accepted	-	-
RT vs RS	0.1695	Accepted	-	-
SA vs LR	0.7691	Accepted	-	-
SA vs RT	0.6565	Accepted	-	-
Execution Time Budget: 60%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	3.43E-15	Rejected	0.5870	Small
LR vs RS	0.7619	Accepted	-	-
RT vs RS	0.0023	Rejected	0.5204	Negligible
SA vs LR	2.84E-13	Rejected	0.5877	Small
SA vs RT	1.22E-09	Rejected	0.5674	Negligible
Execution Time Budget: 70%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	2.16E-23	Rejected	0.6158	Small
LR vs RS	0.9713	Accepted	-	-
RT vs RS	0.0899	Accepted	-	-
SA vs LR	1.25E-21	Rejected	0.6196	Small
SA vs RT	1.65E-18	Rejected	0.6083	Small
Execution Time Budget: 80%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	1.37E-24	Rejected	0.6389	Small
LR vs RS	0.0048	Rejected	0.4694	Negligible
RT vs RS	0.0209	Rejected	0.5283	Negligible
SA vs LR	1.31E-27	Rejected	0.6678	Medium
SA vs RT	4.43E-17	Rejected	0.6141	Small
Execution Time Budget: 90%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	2.80E-29	Rejected	0.6600	Small
LR vs RS	0.4744	Accepted	-	-
RT vs RS	0.1339	Accepted	-	-
SA vs LR	1.51E-28	Rejected	0.6775	Medium
SA vs RT	4.77E-21	Rejected	0.6377	Small
Execution Time Budget: 100%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	8.70E-42	Rejected	0.7108	Medium
LR vs RS	2.11E-05	Rejected	0.5722	Negligible
RT vs RS	1.29E-15	Rejected	0.6073	Small
SA vs LR	1.57E-39	Rejected	0.7092	Medium
SA vs RT	7.91E-32	Rejected	0.6603	Small

Table 5.10: Comparing precision of DTM trained using surrogate-assisted (SA), ML-guided (LR and RT) and random baseline (RS) algorithms with varying execution time budget using Wilcoxon rank sum test [42] and the Vargha-Delaney's $\hat{A}_{12}$ effect size [56] for all the subjects.

Execution Time Budget: 50%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	0.0002	Rejected	0.4497	Negligible
LR vs RS	0.0132	Rejected	0.4724	Negligible
RT vs RS	4.24E-05	Rejected	0.4521	Negligible
SA vs LR	0.1799	Accepted	-	-
SA vs RT	0.5225	Accepted	-	-
Execution Time Budget: 60%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	0.3865	Accepted	-	-Negligible
LR vs RS	0.0003	Rejected	0.4586	Negligible
RT vs RS	1.65E-06	Rejected	0.4499	Negligible
SA vs LR	0.0001	Rejected	0.5488	Negligible
SA vs RT	1.32E-05	Rejected	0.5572	
Execution Time Budget: 70%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	0.0001	Rejected	0.5441	Negligible
LR vs RS	0.0367	Rejected	0.4693	Negligible
RT vs RS	0.0039	Rejected	0.4616	Negligible
SA vs LR	4.37E-09	Rejected	0.5730	Negligible
SA vs RT	7.19E-09	Rejected	0.5814	Small
Execution Time Budget: 80%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	1.90E-07	Rejected	0.5605	Negligible
LR vs RS	0.0001	Rejected	0.4532	Negligible
RT vs RS	0.0014	Rejected	0.4615	Negligible
SA vs LR	4.80E-17	Rejected	0.6077	Small
SA vs RT	6.62E-14	Rejected	0.5972	Small
Execution Time Budget: 90%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	5.57E-12	Rejected	0.5796	Small
LR vs RS	0.0011	Rejected	0.4543	Negligible
RT vs RS	1.11E-06	Rejected	0.4481	Negligible
SA vs LR	3.71E-18	Rejected	0.6260	Small
SA vs RT	8.71E-22	Rejected	0.6298	Small
Execution Time Budget: 100%				
Comparison	p-value	Null Hypothesis	A_{12} estimate	Effect size
SA vs RS	1.16E-13	Rejected	0.5946	Small
LR vs RS	0.0007	Rejected	0.4577	Negligible
RT vs RS	7.80E-07	Rejected	0.4580	Negligible
SA vs LR	5.40E-20	Rejected	0.6357	Small
SA vs RT	5.62E-21	Rejected	0.6296	Small

test results comparing precision of DRM between all possible pairs of SA, RS, LR and RT for different execution time budgets and for all subjects. We can observe that for an execution time budget of 50%, the null hypothesis is accepted for all comparisons which indicates that the distribution of precision over all subjects for all algorithms is equal. The null hypothesis is rejected for time budgets 60% to 100% when the SA vs RS is compared with an effect size varying from Small to Medium. The precision of LR is significantly different from RS for execution budgets 80% and 100% whereas the null hypothesis is accepted in all other cases. The precision of RT is statistically significant for 3 execution time budgets (60%, 80%, and 100%). Finally, the null hypothesis is rejected when the precision of SA is compared with both LR and RT for execution time budgets 60% to 100%. The effect sizes range from Small to Medium for LR and Negligible to Small for RT.

Statistical comparison of Precision for DTM: Table 5.7 presents the Wilcoxon results for the comparison of precision of DTM trained using SA, RS, LR and RT for 15 subjects over varying time budgets. It can be observed that the null hypothesis is accepted for all pairwise comparisons when the algorithms are executed with 50% of the time budget. The null hypothesis is rejected for all the comparisons except SA vs RS with an effect size varying from Negligible when the algorithms are executed with a 60% time budget. Finally, when the time budget is set to 70% to 100%, all the pairwise comparisons are rejected with an effect size varying from Negligible to Small. This indicates that all the algorithms are significantly different from each other in terms of precision.

The detailed accuracy, precision, and recall values for different execution time budgets for each subject are shown in Tables 1 – 12.

The answer to RQ3 is that the dynamic surrogate-assisted test generation algorithm has a significantly higher average Accuracy, Precision, and Recall when compared to ML-guided test generation techniques and Random Search baseline.

5.4 Summary

In this section, we understood the results of the three research questions. The answer to the first research question is that the out of the 8 variants of SA algorithm, SA-DYN is the most optimal algorithm with the best trade-off between accuracy and eficiency. In the second research question, the ML-guided test generation algorithms LR and RT are compared with baseline RS and with one another to identify the algorithm which eficiently generates more boundary tests. It was found that LR generates more boundary tests followed by RT and RS. Finally, the third research question applies the datasets generated using SA, RS, LR, and RT to observe the usefulness of these algorithms in building failure models. Two alternate failure models namely DRM and DTM were implemented and the Accuracy, Precision, and Recall were recorded. The experiment was performed by varying the execution time budget equally for SA, RS, LR, and RT. It was found that, for maximum time budget, SA resulted in higher Accuracy, Precision, and Recall when compared to LR, RT, and RS for both DRM and DTM.

Chapter 6

Related Works

In this chapter, an overview of the previous research work in the area of machine learning for search-based software testing, surrogate-based testing, and building failure models are presented.

6.1 ML for Search-based software testing

Previous research suggests that the application of ML enhances the effectiveness of fuzz testing [45] and Search-based software testing [31]. The application of ML in fuzz testing has proven to be eficient, especially with test sampling, seed generation, and selection of mutation operator [30] [58]. The application of ML in Search-based software testing has shown methods to eficiently and effectively perform software testing on compute-intensive systems such as CPS [41] [5] [32] and autonomous systems [10] [9] [33].

Lee et al. [39] presented a framework to estimate Worst-case Estimation Times (WCET) values of real-time systems at early design stages to ensure that various time-sensitive tasks meet their deadlines. More specifically, the authors attempt to estimate WCET ranges defined by a range of values instead of single discrete values. The proposed approach has two main phases: (1) Finding tasks of real-time systems which miss the deadlines or deliver close to the deadline using a steady-state genetic algorithm. The fitness objective is a quantitative measure defining the degree of deadline misses. Phase 1 creates a dataset representing the task and corresponding time characteristics from which the dataset is labeled into tasks that meet their deadlines and tasks that do not meet their deadlines. (2) The labeled dataset from phase 1 is used to perform supervised learning using logistic regression classifier which infers the WCET ranges. In phase 2, the dataset is first preprocessed by reducing the dimensionality and handling data imbalance. They then refine the logistic regression model iteratively by adding more data points to the labeled dataset using a genetic algorithm and training logistic regression on the dataset. The refinement process continues till the analysis budget is reached. Finally, a "safe" WCET range (i.e., Range of inputs such that the deadline is met) is inferred from the final logistic regression hyperplane that defines the boundary between tasks completed before the deadline and tasks that missed the deadline. In our work, there are two main phases as well. But both phases are geared towards generating test datasets suitable to build failure models. Further, our work experiments with a range of ML models to assist the search-based software testing process.

Feldt et al. [23] experiments on probabilistic programming languages (PPL) and Genetic programming (GP) to improve Search-based software testing. They propose to learn

a generative model rather than searching for an individual or a set of test cases that satisfied requirements under software verification. Additionally, they argue that Search-based software testing requires a large number of execution to search for test cases following specific requirements. Therefore, they suggest that learning generative models and using them for sampling test inputs would be more effective. Another interesting application of Search-based software testing and ML was presented by [26] in which the authors proposed a novel automated technique to identify the conditions (i.e., Assumptions) under which the system under test is guaranteed to satisfy its requirements. The approach begins by generating an initial test suite in which some tests satisfy the requirements while others fail the requirements. They then feed this dataset to a tree-based interpretable machine learning such as a Decision Tree classifier to learn the assumptions (conditions) under which the system under test, when restricted by the inferred conditions, satisfies the requirement. Finally, the assumption is validated by checking if the test cases generated under the constrained environment guarantee to satisfy the requirement. In this work, the authors treat the assumption generation problem as a classification task. Contrary to this, we employ ML models as surrogates to perform a regression task. Therefore, our techniques for ML-assisted test generation experimented with different types of regression machine-learning models as opposed to the classification techniques used in this paper.

In this paper Arrieta et al. [5], authors propose a Search-based software testing approach built based on the well-known multi-objective search algorithm NSGA-II. The goal of the approach is to generate the optimal set of test cases defined by three cost measures. The cost measures represent the requirements coverage, time is taken to execute the test cases to improve the execution cost, and test case similarity to improve the eficiency of the

proposed search technique. On top of that, the authors also defined a crossover operator and three mutation operators to improve the genetic diversity of the samples.

Zhong et al. [62] proposed an evolutionary search algorithm to generate valid failure scenarios for autonomous vehicle simulators. The authors designed a novel seed selection and migration operator to generate temporally sequenced driving scenarios. More specifically, the seed selection for the evolutionary algorithm uses a neural network trained to predict if an input leads to a unique violation. They then rank the inputs based on the confidence of predictions and the top set of inputs are used as the initial population in each iteration. Out of the selected inputs, the inputs with low confidence are mutated using projected gradient descent (PGD) such that the mutated input follows the grammar constraints. The sampling of test cases is conducted iteratively til the budget is reached.

As opposed to Arrieta et al. [5] and Zhong et al. [62], our work does not focus on improving the search algorithms to enhance the software testing technique. Instead, we aimed to investigate the application of ML models in assisting the search-based software testing techniques.

6.2 Surrogate-testing

Dushatskiy et al. [21] suggested a surrogate-assisted evolutionary algorithm for solving optimization problems. Their approach uses Parameterless Population Pyramid (P3) to address non-binary combinatorial optimization problems and ensemble techniques as surrogate models. More specifically, their approach begins by randomly generating and eval-

uating an initial set of test cases. They then train a homogeneous ensemble model (For example, an SVM model with different model configurations creating multiple models) on these test cases. After that, they continue to perform the search but switch to evaluating the test cases using the trained surrogates. Evaluation of test cases using computationally expensive simulators is performed only when the surrogate model returns a high predicted fitness value. The authors' intuition is that if the predicted fitness score is high, then the corresponding test case input should also have a high fitness score when evaluated using a simulator. An important contrast between our surrogate-assisted approach and the one described in this paper is the condition to choose between the actual fitness value from the simulator and the predicted fitness value from the surrogate model. In our approach, we use the surrogate model's error on the test set and check if the predicted fitness score changes its label with two conditions: (1) If the error is added to the predicted fitness value and, (2) If the error is subtracted to the predicted fitness value. As long as the label of the predicted fitness value does not change in both case (1) and case (2), we consider the surrogate's predictions to be reliable. If not, then the test case is passed to the simulator the compute the actual fitness score. The intuition behind this method to check the reliability of the prediction is that the approximation of fitness value near the boundary regions cannot be reliable. Therefore, in our work, the goal of checking the reliability is to ensure that the test inputs with predicted fitness values near the boundary are executed by the CPS. Based on our empirical evaluation, this is an effective strategy because the percentage of incorrectly labeled test inputs over dataset size is on average less than 2%. Another difference between our approach and this paper is in the number of surrogate models considered for building an ensemble model. In our work, we considered seven dif-

ferent machine learning models as opposed to four machine learning models used in this paper.

Haq et al. [29] proposed Surrogate-Assisted Many-Objective Test suite generation Algorithm (SAMOTA) for test suite generation for DNN-enabled systems to incorporate surrogate − testing with multi-objective search algorithms. The research outcome of the paper was to solve three main challenges in online testing of the system under test: (1) Testing multiple requirements of the system under test simultaneously (2) Testing simulators that are computationally expensive (3) Dealing with large search spaces which cannot be exhaustively explored. The approaches proposed in this thesis aims to solve the second and third challenge as well. SAMOTA operates in two stages. The algorithm starts by performing a global search using the trained surrogate models to explore the search space. More specifically, the global search returns a set of best test cases defined by predicted fitness and another set of test cases that are the most uncertain in terms of whether the test case is failure-inducing or not. The uncertainty is calculated based on the level of disagreement among the surrogate models considered. They then perform a local search with a genetic algorithm to exploit promising areas around the best test cases from the global search. The test cases that are found to be in cooperation with both global and local search are simulated and these extra simulations are also used to refine the surrogate models. In their work, the prediction of the ensemble model is an average of three surrogate models namely polynomial regression, Kriging, and radial basis function network. However, in our work, we create a dynamic ensemble model using seven different machine-learning models. Unlike their ensemble model, our ensemble technique chooses the best prediction rather than the average of predictions. Our experiment results showed that choosing the

best prediction leads to more accurate results than choosing an average prediction in the context of CPS.

Software testing for compute-intensive CPS is especially dificult because eficient testing requires evaluating a large number of test cases which may not be feasible due to the systems' non-linear mathematics for execution. In the paper, Menghi et al. [43] propose a technique to enable eficient black-box software testing for compute-intensive CPS. To do so, the authors introduce AppRoxImation-based TEst generatiOn (ARIsTEO) which involves building a surrogate model which can be executed faster than approximates the CPS. Unlike machine-learning models used as surrogate models in our work, the authors in this paper have used System Identification which allows estimating the system under test based on a few input-output data to create an initial surrogate model. They then iteratively perform black-box testing using the surrogate model until a failure-inducing test case is found. The approach then checks if the found failure test case is spurious or not. To do so, the failure test case is passed to the system under test to check if the test case results in a failure situation. If the test case was spurious, the surrogate model is retrained using the spurious failure-inducing test case. A main difference between our work and their work is that their work investigates using System Identification as surrogate models as opposed to ML models as shown in our work.

Matinnejad et al. [41] extended their previous work on meta-heuristic search algorithms to perform automated testing of continuous controllers with fixed configurations. Specifically, the authors focused on improving the support to all feasible configurations of continuous controllers which lead to an increase in the size of search space. To handle that, the approach starts by performing dimensionality reduction to identify and consider

input features with significant impact on the output of the objective function used i.e., Fitness score. In our work, the intuition behind not performing dimensionality reduction and feature selection is that all the input signals of Simulink should be represented in building failure models. This way, there may be chances of the failure models generating rules that characterize the hidden patterns of failure-inducing test inputs. Subsequently, an exploratory random search is performed on the reduced dataset to identify areas in which it is likely to find failure-inducing test cases and then use a regression tree to divide the search space into multiple partitions. They then perform a surrogate-assisted single-state search in these partitions of search space to find the worst-case scenarios.

Abdessalem et al. [10] proposed a meta-heuristic approach assisted by neural networks to perform system-testing on physics-based simulation platforms for the Pedestrian Detection Vision-based (PeVi) system. In their work, Non-dominated Sorting Genetic Algorithm (NSGAII) is implemented in which a partial order relation rank is computed based on the quantitative fitness measure to select the best individuals (i.e., Input test cases). The search task is considered multi-objective due to multiple requirements that need to be satisfied for the system to work without any failures. More specifically, four different functions that measure various metrics that could impact the possibility of a collision are addressed. This approach is further improved by introducing a neural network model as a surrogate to predict the fitness measure. More specifically, the predicted fitness score is used to calculate the partial order relation rank instead of the actual fitness score which is calculated by executing the simulator.

Beglerovic et al. [9] attempted to improve the testing methods of autonomous vehicles. The challenges that were addressed in this paper are a large number of test scenarios,

large parameter space, and computationally expensive simulation runs. To address these issues, the authors proposed surrogate-assisted testing with stochastic optimization. The approach begins by generating an initial set of test inputs and evaluating their cost function by simulating the test inputs. The cost function returns a numerical output based on the behavior of the system. The approach then builds surrogate models (specifically Kriging models) that act as an approximation model to the autonomous driving system under test. In this thesis, apart from Krigig, we also evaluate the performance of six other ML models as surrogate models from the literature as previously shown in 3.2. The generated test cases are evaluated for the number of iterations defined to find a faulty behavior. If the number of iterations is exceeded, evaluation of the cost function is performed using the surrogate model. The numerical output from the surrogate model is then used to perform stochastic optimization using Differential Evolution and Particle Swarm Optimization to find the search space where the global cost function is minimum. The search space is zoomed in and then the test cases are generated within the restricted space. This method proceeds in this iterative way until the testing budget is reached.

Arrieta [3] introduce digital twins and proposed multi-fidelity digital twins to reduce the computation of software testing. At a high level, the authors propose to create assumptions to simplify the simulators such that only the subsystem that is under the scope of testing interest comprises the actual modules, and all the other components are replaced by a simplified digital twin that is relatively cheaper to execute. The authors define "multi-fidelity digital twins" - as multiple digital twins created with different levels of assumptions within the scopes of testing interests. This paper hypothesizes that Multi-fidelity digital twins improve CPS testing techniques. The authors then address three research aspects of

multi-fidelity digital twins. First, the authors discuss the methods by which digital twins can be quantitatively measured. They then discuss methods (such as search algorithms or artificial intelligence) to automate the generation and maintenance of multi-fidelity digital twins. Lastly, they also discuss multiple software verification and validation tasks (such as test generation, debugging, and test oracle generation) that can be further improved by multi-fidelity digital twins. Our work also aims to improve the automated testing strategies of digital twins representing CPS. Moreover, as per the author's recommendation, our approaches implement different search algorithms and Machine learning models that assist the software testing framework.

Wang et al. [59] proposed an approach to identify boundary-defining safety-critical scenarios in the context of Autonomous Vehicle systems. The approach involves a surrogate model approximating the autonomous vehicle system under test with a gradient descent search algorithm to eficiently search for the boundary defining the area of safety-critical scenarios. The proposed technique first takes the input parameter search space as input to search and generate scenarios using adaptive sampling. The test scenarios generated using the sampling technique are further used to train a Multi-layered Perceptron (MLP) which acts as a surrogate model for the system under test. The technique then searches for safety-critical scenarios through gradient descent algorithms namely the basic Steepest Descent (SD) algorithm, Gradient Descent with Momentum (GDM) algorithm, and Adaptive Moment Estimation (Adam) algorithm.

6.3 Building Failure Models

Recent advancements in the field of software testing for CPS suggest a shift from generating and executing a limited number of test cases to learning failure models that can characterize system failures. [23] [27]. The research work conducted in this thesis derives a strong motivation from these observations and therefore, apart from building ML-assisted test generation frameworks, our goal is to incorporate failure models to learn the hidden patterns characterizing failure circumstances of the system. Learning failure models is essential to eficiently perform software testing as it allows the engineers to identify the root cause of a failure situation in the system under test. Although software verification techniques generate and perform testing on inputs by fuzzing (i.e., Random tester), the failure-inducing inputs are observed independently as opposed to building failure models to identify a set of failure-inducing inputs at a system level that can potentially lead to global explanations to the failure situation of the system under test. A global explanation allows for a precise fix (i.e., a fix that handles only the failure-inducing inputs without altering the performance of the system on the non-failure-inducing systems).

Kampmann et al. [35] proposed an automated approach called ALHAZEN to build failure models that precisely explain the failure circumstances of a computer program. The approach comprises four steps: (1) Using a grammar to parse the inputs into individual elements. (2) Training a decision tree classifier to learn the association of input elements with the behavior of the system under test. (3) The grammar created in (1) is further used to generate additional inputs to either support or disprove a possible explanation for the failure circumstance. (4) involves iteratively performing Steps (2) and (3) to refine the

decision tree that captures the information to explain the failure circumstance of a program. The authors in this paper aim to learn failure models for string-based inputs as opposed to continuous and discrete numerical inputs in our work. Specifically, our work focuses on CPS as the system under test, unlike program code which is the system under test in this paper. Another distinction to be noted is that, unlike their approach which uses a discrete labeling approach, all our proposed frameworks deal with a numerical quantitative fitness value that defines the severity of the failure.

Gopinath et al. [27] presented a technique to characterize failure-inducing inputs for computer programs by creating abstract failure-inducing input that abstracts over the original input. The approach starts by identifying and characterizing a single failure-inducing input. Using the input grammar which is provided as an input to the technique, the failure-inducing input is first converted into a minimal failure-inducing input using syntax-based reduction. They then replace the elements of the input with abstract elements from the input grammar in a process termed abstraction. Further, they isolate all the independent expressions and expressions that are shared with other expressions. Finally, any lexical tokens such as white spaces and comments are removed with the assumption that lexical terms are not useful while debugging (i.e. Learning the failure of the program code). According to the authors, the abstract failure-inducing model can be used to diagnose failure circumstances and generate additional failure-inducing inputs to validate a fix to a failure circumstance. This automated approach attempts to develop a generalized failure-inducing input that can further generate more failure-inducing inputs to identify a set of failure-inducing inputs. Our work and this paper share the same motivation of building failure models to understand the general pattern of failure-inducing inputs. But in our

work, we focus on developing eficient testing strategies to test compute-intensive CPS as opposed to software programs in their work. Therefore, the test inputs in our work primarily deal with numerical values while their work deals with a set of strings representing a program.

Chapter 7

Conclusions

In this thesis, we proposed a framework for two alternate types of ML-assisted test generation for CPS models to improve the eficiency and accuracy of existing search-based software testing techniques. Additionally, a module to build failure models using the data generated by different ML-test generation approaches were introduced. As a result, the data generated by the ML-assisted test generation approaches were used to train failure models that were able to effectively characterize the failure circumstances of the system under test at a system level. Recall that the proposed ML-assisted test generation has two main stages: (1) Preprocessing and (2) Main Loop. In the main loop, a module to estimate the reliability of each prediction for a given test input was also designed. To our knowledge, the design of dynamic surrogate-assisted test generation, the method to estimate the reliability of ML prediction, and the incorporation of failure models to characterize failure inputs of the system under test are the main novel aspects of our work.

7.1 Research Contributions

The following research contributions were made in this thesis:

- Comparison of different surrogate-assisted test generation approaches for testing CPS models: In the case of surrogate-assisted test generation, the task of the ML models was to approximate the CPS model and predict the fitness value for a given test input. Since the target feature is continuous, we framed a regression task. Therefore, a total of 7 ML models in the literature were identified. We then developed 8 different versions of surrogate-assisted test generation approaches. Out of which, 7 versions involved the application of a fixed ML model for prediction and 1 version involved an ensemble system of 7 ML models for prediction. An elaborate comparison was performed to understand the accuracy of the predictions and the eficiency of all the versions. As a result of this experimentation, we were able to conclude that the surrogate-assisted test generation with the dynamic ensemble for prediction (i.e., SA-DYN) provides the best trade-off between the eficiency of the dataset and accuracy by generating test datasets that are at least 28% more accurate and at least 33% larger.

- Comparison of different ML-guided test generation approaches for testing CPS models: In the case of ML-guided test generation, the task of ML models was to guide the sampling towards the boundary region that differentiates the pass and fail test inputs. Two explainable ML models were implemented to extract information about the dataset and identify the test inputs that were most probably near the

boundary. An elaborate study was conducted to understand if the ML-guided test generation were successful in generating test inputs near the boundary. Based on the experimentation, we can conclude that ML-guided test generation techniques were able to generate at least 14.5% more test inputs near the boundary.

- Comparison of surrogate-assisted, ML-guided and Random Search test generation for testing CPS models based on the performance of failure models: The datasets generated from different ML-assisted test generation and Random Search test generation were implemented to train failure models to understand the relationship between the eficiency of failure models and the approach (i.e., dataset) used to train the failure models. An extensive comparison was performed to compare different approaches based on the Accuracy, Precision, and Recall of the failure models. Based on the results from this experimentation, we can show that as the time budget is increased, SA-DYN surrogate-assisted test generation results in the highest improved Accuracy, Precision, and Recall.

Overall, when the time budget is set to maximum, the SA-DYN surrogate-assisted test generation algorithm generates 2.6 times larger datasets when compared to the other alternatives. About 98.6% of test labels were correctly labeled by SA-DYN and we were also able to prove that the small number of incorrect predictions did not impact the accuracy of the failure model when compared with the other alternatives. Therefore, despite generating larger datasets, the accuracy of the predictions was not significantly compromised. This allows SA-DYN to build accurate failure models indicating that these models can infer accurate rules that represent the failures in computationally intensive systems.

7.2 Future Work

This piece of research definitely can be extended further in multiple aspects. Firstly, the ML-guided and surrogate-assisted test generation were contested against each other in this thesis. There may be a possibility to obtain the best of both surrogate-assisted and ML-guided test generation. This aspect is interesting and can be explored in the future. Secondly, although an extensive comparison of the performance of failure models was conducted by varying the execution time budget, there is still scope for understanding the impact of varying the time budget over a much larger time budget. In our work, due to the computationally expensive nature of the Simulators, the maximum time budget was set to 600 in terms of the number of executions of the simulator. A separate study can be conducted to understand the impact of much higher time budgets to understand the performance of surrogate-assisted and ML-guided test generation algorithms. Finally, in this thesis, 5 different CPS benchmarks were used for evaluation. There is a scope to apply the proposed framework with a broader class of systems to fully understand the performance of these algorithms over systems from a variety of domains.

References

[1] SIMULINK definition. https://www.mathworks.com/products/simulink.html. (Accessed: November 2022).

[2] Logistic regression. http://faculty.cas.usf.edu/mbrannick/regression/Logistic.html, (Accessed: January 2023).

[3] Aitor Arrieta. Multi-fidelity digital twins: a means for better cyber-physical systems testing? arXiv, abs/2101.05697, 2021.

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

[5] Aitor Arrieta, Shuai Wang, Urtzi Markiegi, Goiuria Sagardui, and Leire Etxeberria. Search-based test case generation for cyber-physical systems. In 2017 IEEE Congress on Evolutionary Computation (CEC), pages 688–697. IEEE, 2017.

[6] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In NDSS, 2019.

[7] Radhakisan Sohanlal Baheti and Helen Gill. Cyber-physical systems. 2019 IEEE International Conference on Mechatronics (ICM), 2019.

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

[9] Halil Beglerovic, Michael Stolz, and Martin Horn. Testing of autonomous vehicles using surrogate models and stochastic optimization. In 2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC), pages 1–6. IEEE, 2017.

[10] Raja Ben Abdessalem, Shiva Nejati, Lionel C Briand, and Thomas Stifter. 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, pages 63–74, 2016.

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

[12] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. J. Mach. Learn. Res., 13:281–305, 2012.

[13] L. Breiman. Bagging predictors. Machine Learning, 24:123–140, 2004.

[14] L. Breiman, Jerome H. Friedman, Richard A. Olshen, and C. J. Stone. Classification and regression trees. 1984.

[15] Leo Breiman. Random Forests. Machine Learning, 45(1):5–32, 2001.

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

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

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

[19] Doriana Marilena D'Addona. Neural Network, pages 911–918. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

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

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

[22] Rong-En Fan, Pai-Hsuen Chen, and Chih-Jen Lin. Working set selection using second order information for training support vector machines. J. Mach. Learn. Res., 6:1889–1918, 2005.

[23] Robert Feldt and Shin Yoo. Flexible probabilistic modeling for search based test data generation. In Proceedings of the 13th International Workshop on Search-Based Software Testing (SBST), pages 537–540, 2020.

[24] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In Future of Software Engineering (FOSE '07), pages 37–54, 2007.

[25] P. Frazier. A tutorial on bayesian optimization. ArXiv, abs/1807.02811, 2018.

[26] Khouloud Gaaloul, Claudio Menghi, Shiva Nejati, Lionel C Briand, and David Wolfe. 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, pages 159–171, 2020.

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

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

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

[30] Mark Harman, Sung Gon Kim, Kiran Lakhotia, Phil McMinn, and Shin Yoo. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In 2010 Third International Conference on Software Testing, Verification, and Validation Workshops, pages 182–191. IEEE, 2010.

[31] Mark Harman and Phil McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. IEEE Transactions on Software Engineering, 36(2):226–247, 2009.

[32] Dmytro Humeniuk, Giuliano Antoniol, and Foutse Khomh. Data driven testing of cyber physical systems. In 2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST), pages 16–19. IEEE, 2021.

[33] Dmytro Humeniuk, Foutse Khomh, and Giuliano Antoniol. A search-based framework for automatic generation of testing environments for cyber-physical systems. Information and Software Technology, page 106936, 2022.

[34] Jeff C. Jensen, Danica H. Chang, and Edward A. Lee. A model-based design methodology for cyber-physical systems. In 2011 7th International Wireless Communications and Mobile Computing Conference, pages 1666–1671, 2011.

[35] Alexander Kampmann, Nikolas Havrikov, Ezekiel O Soremekun, and Andreas Zeller. 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, pages 1228–1239, 2020.

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

[37] Neil Kulkarni, Caroline Lemieux, and Koushik Sen. Learning highly recursive input grammars. In 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 456–467. IEEE, 2021.

[38] Edward A. Lee. Cyber physical systems: Design challenges. In 2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC), pages 363–369, 2008.

[39] Jaekwon Lee, Seung Yeob Shin, Shiva Nejati, Lionel C Briand, and Yago Isasi Parache. Estimating probabilistic safe wcet ranges of real-time systems at design stages. ACM Transactions on Software Engineering and Methodology, 2022.

[40] Sean Luke. Essentials of Metaheuristics. Lulu, second edition, 2013.

[41] Reza Matinnejad, Shiva Nejati, Lionel Briand, and Thomas Brcukmann. 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, pages 163–174, 2014.

[42] Patrick E McKnight and Julius Najab. Mann-whitney u test. The Corsini encyclopedia of psychology, pages 1–1, 2010.

[43] Claudio Menghi, Shiva Nejati, Lionel Briand, and Yago Isasi Parache. 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), pages 372–384. IEEE, 2020.

[44] Claudio Menghi, Shiva Nejati, Khouloud Gaaloul, and Lionel C Briand. 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, pages 27–38, 2019.

[45] Barton P Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. Communications of the ACM, 33(12):32–44, 1990.

[46] Melanie Mitchell, John H. Holland, and Stephanie Forrest. When will a genetic algorithm outperform hill climbing? NIPS'93, page 51–58, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

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

[48] Novi Quadrianto, Kristian Kersting, and Zhao Xu. Gaussian Process, pages 428–439. Springer US, Boston, MA, 2010.

[49] J. R. Quinlan. Induction of decision trees. Machine Learning, 1:81–106, 1986.

[50] David Rogers. Random search and insect population models. Journal of Animal Ecology, 41(2):369–383, 1972.

[51] Lior Rokach and Oded Maimon. Decision Trees, pages 165–192. Springer US, Boston, MA, 2005.

[52] Jianhua Shi, Jiafu Wan, Hehua Yan, and Hui Suo. A survey of cyber-physical systems. In 2011 International Conference on Wireless Communications and Signal Processing (WCSP), pages 1–6, 2011.

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

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

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

[56] András Vargha and Harold D Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. Journal of Educational and Behavioral Statistics, 25(2):101–132, 2000.

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

[58] Yan Wang, Peng Jia, Luping Liu, Cheng Huang, and Zhonglin Liu. A systematic review of fuzzing based on machine learning techniques. PloS one, 15(8):e0237749, 2020.

[59] Yiyun Wang, Rongjie Yu, Shuhan Qiu, Jian Sun, and Haneen Farah. Safety performance boundary identification of highly automated vehicles: A surrogate model-based gradient descent searching approach. IEEE Transactions on Intelligent Transportation Systems, 23(12):23809–23820, 2022.

[60] Ian H. Witten, Eibe Frank, and Mark A. Hall. Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, Amsterdam, 3 edition, 2011.

[61] Xi Zheng, Christine Julien, Miryung Kim, and Sarfraz Khurshid. Perceptions on the state of the art in verification and validation in cyber-physical systems. IEEE Systems Journal, 11(4):2614–2627, 2017.

[62] Ziyuan Zhong, Gail E. Kaiser, and Baishakhi Ray. Neural network guided evolutionary fuzzing for finding trafic violations of autonomous vehicles. CoRR, abs/2109.06126, 2021.

A P P E N D I C E S

Table 1: Average accuracy, recall and precision of DRM over all the runs of surrogate-assisted (SA), ML-guided (LR and RT) and random baseline (RS) for all 15 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
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
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
Average	0.55	0.54	0.55	0.55

Table 2: Average accuracy, recall and precision of DRM over all the runs of surrogate-assisted (SA), ML-guided (LR and RT) and random baseline (RS) for all 15 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
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
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
Average	0.61	0.54	0.56	0.57

Table 3: 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
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
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
Average	0.63	0.55	0.57	0.58

Table 4: Average accuracy, recall and precision of DRM 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
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
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
Average	0.65	0.56	0.57	0.59

Table 5: Average accuracy, recall and precision over all the runs of surrogate-assisted (SA), ML-guided (LR and RT) and random baseline (RS) for all 15 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
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
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
Average	0.66	0.57	0.58	0.60

Table 6: Average accuracy, recall and precision of DRM over all the runs of surrogate-assisted (SA), ML-guided (LR and RT) and random baseline (RS) for all 15 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
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
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
Average	0.72	0.57	0.62	0.64

Table 7: Average accuracy, recall and precision of DTM over all the runs of surrogate-assisted (SA), ML-guided (LR and RT) and random baseline (RS) for all 15 subjects with 50% execution time budget.

Metric: Accuracy				
Subject	SA	RS	LR	RT
TU1	0.81	0.80	0.80	0.80
TU2	0.95	0.93	0.96	0.95
TU3	0.86	0.83	0.85	0.85
TU4	0.86	0.86	0.86	0.87
TU5	0.85	0.85	0.88	0.86
TU6	0.85	0.84	0.85	0.86
TU7	0.80	0.77	0.81	0.80
TU8	0.97	0.95	0.97	0.97
TU9	0.98	0.96	0.97	0.97
REG	0.73	0.71	0.73	0.71
AP1	0.92	0.93	0.93	0.91
AP2	0.75	0.73	0.73	0.74
AP3	0.66	0.65	0.68	0.68
Average	0.84	0.83	0.85	0.84
Metric: Recall				
Subject	SA	RS	LR	RT
TU1	0.42	0.45	0.46	0.42
TU2	0.80	0.83	0.77	0.77
TU3	0.73	0.77	0.73	0.76
TU4	0.71	0.72	0.76	0.74
TU5	0.79	0.79	0.77	0.80
TU6	0.80	0.82	0.81	0.80
TU7	0.79	0.75	0.78	0.80
TU8	0.98	0.98	0.97	0.97
TU9	0.99	0.99	0.98	0.98
REG	0.73	0.74	0.74	0.72
AP1	0.96	0.97	0.97	0.94
AP2	0.42	0.34	0.39	0.43
AP3	0.57	0.56	0.56	0.57
Average	0.75	0.75	0.75	0.75
Metric: Precision				
Subject	SA	RS	LR	RT
TU1	0.54	0.61	0.55	0.51
TU2	0.82	0.85	0.82	0.81
TU3	0.75	0.74	0.66	0.67
TU4	0.76	0.75	0.74	0.69
TU5	0.79	0.79	0.75	0.75
TU6	0.78	0.77	0.78	0.77
TU7	0.75	0.73	0.77	0.75
TU8	0.97	0.98	0.97	0.98
TU9	0.97	0.97	0.97	0.96
REG	0.71	0.71	0.71	0.69
AP1	0.93	0.93	0.93	0.92
AP2	0.62	0.58	0.58	0.58
AP3	0.53	0.53	0.54	0.55
Average	0.76	0.76	0.75	0.74

Table 8: Average accuracy, recall and precision of DTM over all the runs of surrogate-assisted (SA), ML-guided (LR and RT) and random baseline (RS) for all 15 subjects with 60% execution time budget.

Metric: Accuracy				
Subject	SA	RS	LR	RT
TU1	0.97	0.97	0.95	0.96
TU2	0.88	0.88	0.87	0.86
TU3	0.90	0.88	0.87	0.88
TU4	0.90	0.90	0.88	0.87
TU5	0.89	0.87	0.88	0.85
TU6	0.83	0.83	0.80	0.82
TU7	0.83	0.81	0.81	0.80
TU8	0.98	0.98	0.98	0.98
TU9	0.98	0.98	0.98	0.98
REG	0.73	0.75	0.74	0.72
AP1	0.94	0.93	0.90	0.93
AP2	0.74	0.75	0.74	0.73
AP3	0.68	0.67	0.69	0.67
Average	0.86	0.86	0.85	0.85
Metric: Recall				
Subject	SA	RS	LR	RT
TU1	0.57	0.52	0.47	0.40
TU2	0.80	0.81	0.82	0.80
TU3	0.78	0.75	0.75	0.76
TU4	0.77	0.75	0.74	0.72
TU5	0.81	0.77	0.80	0.79
TU6	0.84	0.83	0.83	0.82
TU7	0.82	0.78	0.81	0.80
TU8	0.99	0.96	0.98	0.98
TU9	0.99	0.96	0.99	0.99
REG	0.71	0.70	0.73	0.75
AP1	0.98	0.91	0.97	0.96
AP2	0.37	0.41	0.38	0.40
AP3	0.59	0.51	0.55	0.55
Average	0.77	0.74	0.75	0.75
Metric: Precision				
Subject	SA	RS	LR	RT
TU1	0.71	0.67	0.48	0.54
TU2	0.86	0.83	0.83	0.83
TU3	0.75	0.72	0.69	0.71
TU4	0.78	0.71	0.75	0.72
TU5	0.81	0.73	0.80	0.73
TU6	0.80	0.78	0.77	0.79
TU7	0.79	0.73	0.77	0.74
TU8	0.97	0.97	0.98	0.98
TU9	0.97	0.97	0.97	0.97
REG	0.73	0.70	0.73	0.70
AP1	0.93	0.88	0.88	0.93
AP2	0.61	0.61	0.64	0.60
AP3	0.55	0.54	0.57	0.53
Average	0.79	0.76	0.76	0.75

Table 9: Average accuracy, recall and precision of DTM over all the runs of surrogate-assisted (SA), ML-guided (LR and RT) and random baseline (RS) for all 15 subjects with 70% execution time budget.

Metric: Accuracy				
Subject	SA	RS	LR	RT
TU1	0.98	0.97	0.96	0.96
TU2	0.90	0.88	0.88	0.88
TU3	0.90	0.89	0.87	0.89
TU4	0.92	0.90	0.90	0.89
TU5	0.90	0.88	0.88	0.86
TU6	0.85	0.85	0.86	0.83
TU7	0.85	0.83	0.83	0.82
TU8	0.97	0.98	0.97	0.98
TU9	0.98	0.98	0.98	0.98
REG	0.75	0.76	0.73	0.75
AP1	0.94	0.93	0.89	0.94
AP2	0.74	0.74	0.74	0.77
AP3	0.70	0.68	0.70	0.67
Average	0.88	0.87	0.86	0.86
Metric: Recall				
Subject	SA	RS	LR	RT
TU1	0.63	0.51	0.51	0.52
TU2	0.83	0.82	0.83	0.82
TU3	0.79	0.75	0.76	0.77
TU4	0.79	0.77	0.80	0.77
TU5	0.84	0.80	0.81	0.78
TU6	0.83	0.84	0.87	0.83
TU7	0.85	0.84	0.82	0.81
TU8	0.99	0.98	0.97	0.98
TU9	0.99	0.99	0.99	0.99
REG	0.70	0.77	0.72	0.75
AP1	0.98	0.96	0.96	0.97
AP2	0.41	0.41	0.38	0.48
AP3	0.63	0.59	0.57	0.55
Average	0.79	0.77	0.77	0.77
Metric: Precision				
Subject	SA	RS	LR	RT
TU1	0.76	0.66	0.52	0.59
TU2	0.89	0.84	0.84	0.85
TU3	0.77	0.75	0.69	0.72
TU4	0.85	0.77	0.79	0.75
TU5	0.82	0.80	0.79	0.76
TU6	0.83	0.83	0.84	0.80
TU7	0.81	0.78	0.80	0.77
TU8	0.97	0.98	0.98	0.98
TU9	0.97	0.98	0.97	0.97
REG	0.77	0.74	0.72	0.73
AP1	0.93	0.94	0.88	0.94
AP2	0.60	0.62	0.61	0.66
AP3	0.57	0.55	0.57	0.53
Average	0.81	0.79	0.77	0.77

Table 10: Average accuracy, recall and precision of DTM over all the runs of surrogate-assisted (SA), ML-guided (LR and RT) and random baseline (RS) for all 15 subjects with 80% execution time budget.

Metric: Accuracy				
Subject	SA	RS	LR	RT
TU1	0.98	0.85	0.97	0.96
TU2	0.90	0.86	0.88	0.89
TU3	0.91	0.82	0.90	0.89
TU4	0.93	0.85	0.90	0.89
TU5	0.90	0.87	0.89	0.88
TU6	0.88	0.85	0.87	0.86
TU7	0.87	0.84	0.84	0.84
TU8	0.98	0.95	0.98	0.98
TU9	0.98	0.98	0.98	0.99
REG	0.75	0.75	0.72	0.75
AP1	0.96	0.86	0.90	0.92
AP2	0.77	0.76	0.74	0.75
AP3	0.71	0.69	0.71	0.69
Average	0.89	0.84	0.87	0.87
Metric: Recall				
Subject	SA	RS	LR	RT
TU1	0.70	0.52	0.60	0.61
TU2	0.83	0.82	0.83	0.82
TU3	0.80	0.80	0.79	0.77
TU4	0.83	0.80	0.77	0.74
TU5	0.83	0.82	0.82	0.81
TU6	0.88	0.87	0.89	0.87
TU7	0.86	0.84	0.84	0.84
TU8	0.99	0.88	0.98	0.98
TU9	0.99	0.89	0.99	0.99
REG	0.69	0.66	0.71	0.77
AP1	0.99	0.88	0.97	0.97
AP2	0.49	0.42	0.37	0.42
AP3	0.65	0.56	0.59	0.61
Average	0.81	0.75	0.78	0.78
Metric: Precision				
Subject	SA	RS	LR	RT
TU1	0.82	0.63	0.64	0.62
TU2	0.91	0.89	0.85	0.87
TU3	0.80	0.76	0.74	0.74
TU4	0.85	0.82	0.78	0.77
TU5	0.84	0.81	0.80	0.79
TU6	0.86	0.81	0.84	0.84
TU7	0.84	0.79	0.80	0.80
TU8	0.97	0.97	0.98	0.98
TU9	0.97	0.96	0.98	0.98
REG	0.78	0.75	0.71	0.73
AP1	0.95	0.92	0.89	0.91
AP2	0.67	0.66	0.61	0.64
AP3	0.58	0.51	0.58	0.56
Average	0.83	0.79	0.79	0.79

Table 11: Average accuracy, recall and precision of DTM over all the runs of surrogate-assisted (SA), ML-guided (LR and RT) and random baseline (RS) for all 15 subjects with 90% execution time budget.

Metric: Accuracy				
Subject	SA	RS	LR	RT
TU1	0.98	0.92	0.97	0.97
TU2	0.92	0.85	0.89	0.89
TU3	0.92	0.90	0.89	0.89
TU4	0.94	0.83	0.91	0.90
TU5	0.92	0.82	0.89	0.87
TU6	0.87	0.88	0.86	0.84
TU7	0.89	0.83	0.86	0.84
TU8	0.98	0.98	0.98	0.98
TU9	0.98	0.99	0.98	0.99
REG	0.76	0.74	0.74	0.76
AP1	0.97	0.93	0.88	0.93
AP2	0.78	0.74	0.75	0.75
AP3	0.73	0.69	0.71	0.68
Average	0.89	0.85	0.87	0.87
Metric: Recall				
Subject	SA	RS	LR	RT
TU1	0.72	0.56	0.62	0.62
TU2	0.86	0.84	0.84	0.84
TU3	0.81	0.78	0.78	0.76
TU4	0.83	0.77	0.81	0.76
TU5	0.86	0.82	0.84	0.81
TU6	0.87	0.87	0.87	0.84
TU7	0.89	0.83	0.86	0.86
TU8	0.99	0.98	0.98	0.98
TU9	0.99	0.99	0.99	0.99
REG	0.70	0.78	0.74	0.78
AP1	0.99	0.88	0.96	0.97
AP2	0.51	0.34	0.41	0.47
AP3	0.67	0.48	0.58	0.59
Average	0.82	0.76	0.79	0.79
Metric: Precision				
Subject	SA	RS	LR	RT
TU1	0.83	0.71	0.60	0.62
TU2	0.91	0.87	0.86	0.85
TU3	0.83	0.76	0.74	0.73
TU4	0.88	0.81	0.79	0.79
TU5	0.86	0.81	0.81	0.77
TU6	0.86	0.86	0.84	0.82
TU7	0.86	0.82	0.82	0.79
TU8	0.97	0.98	0.98	0.98
TU9	0.97	0.98	0.98	0.98
REG	0.78	0.74	0.73	0.73
AP1	0.96	0.95	0.87	0.93
AP2	0.68	0.65	0.62	0.63
AP3	0.60	0.56	0.59	0.55
Average	0.85	0.81	0.79	0.78

Table 12: Average accuracy, recall and precision of DTM over all the runs of surrogate-assisted (SA), ML-guided (LR and RT) and random baseline (RS) for all 15 subjects with 100% execution time budget.

Metric: Accuracy				
Subject	SA	RS	LR	RT
TU1	0.98	0.97	0.97	0.96
TU2	0.92	0.81	0.90	0.89
TU3	0.92	0.83	0.90	0.90
TU4	0.94	0.86	0.91	0.90
TU5	0.92	0.86	0.90	0.87
TU6	0.88	0.83	0.87	0.87
TU7	0.89	0.84	0.85	0.84
TU8	0.98	0.96	0.98	0.98
TU9	0.98	0.94	0.99	0.99
REG	0.77	0.76	0.72	0.77
AP1	0.88	0.76	0.75	0.76
AP2	0.72	0.70	0.71	0.69
AP3	0.97	0.92	0.88	0.94
NLG	0.85	0.77	0.76	0.75
FSM	0.94	0.85	0.85	0.83
Average	0.90	0.84	0.86	0.86
Metric: Recall				
Subject	SA	RS	LR	RT
TU1	0.68	0.57	0.67	0.59
TU2	0.87	0.86	0.86	0.83
TU3	0.82	0.81	0.80	0.79
TU4	0.86	0.81	0.80	0.79
TU5	0.86	0.84	0.85	0.80
TU6	0.88	0.90	0.87	0.87
TU7	0.89	0.83	0.85	0.84
TU8	0.99	0.95	0.98	0.98
TU9	0.99	0.96	0.99	0.99
REG	0.70	0.78	0.70	0.78
AP1	0.53	0.45	0.41	0.47
AP2	0.63	0.61	0.59	0.60
AP3	0.99	0.94	0.96	0.97
NLG	0.90	0.86	0.85	0.84
FSM	0.87	0.68	0.70	0.68
Average	0.83	0.79	0.79	0.79
Metric: Precision				
Subject	SA	RS	LR	RT
TU1	0.80	0.66	0.67	0.60
TU2	0.92	0.88	0.86	0.87
TU3	0.82	0.77	0.75	0.75
TU4	0.87	0.82	0.82	0.78
TU5	0.88	0.82	0.82	0.79
TU6	0.87	0.87	0.86	0.85
TU7	0.86	0.80	0.81	0.79
TU8	0.98	0.98	0.98	0.98
TU9	0.97	0.98	0.98	0.99
REG	0.80	0.75	0.72	0.76
AP3	0.96	0.96	0.88	0.94
AP1	0.69	0.66	0.64	0.65
AP2	0.59	0.58	0.59	0.55
NLG	0.83	0.85	0.85	0.85
FSM	0.84	0.70	0.59	0.55
Average	0.85	0.81	0.79	0.78