A Test Input Generation Framework based on Data-Driven Models for Cyber-Physical Systems

Linh Ngoc Le[1], Maximilian Schmidt[1], M.Sc. and Prof. Goerschwin Fey[1], Dr.-Ing.

[1]*Hamburg University of Technology, Hamburg, Germany*

Abstract

Testing Cyber-Physical Systems (CPSs) is essential to ensure their reliability and safety. A common approach in this field is model-based testing, as it allows the systematic generation of test inputs from system models and supports early fault detection at reduced cost. However, constructing these models is often difficult and time-consuming due to the inherent complexity of CPSs [1]. To address this, data-driven models can be employed within model-based testing to reduce the manual effort and domain expertise required to build system models. This paper presents a framework for test input generation based on data-driven models for CPSs. The framework constructs a decision tree surrogate from the data-driven model to guide input generation. By analysing the tree's internal structure, the framework partitions the input domain of the system under test into equivalence classes. Test inputs are then sampled statistically from each class based on user-defined requirements. Experimental results show that the framework can produce challenging test inputs that span a wide range of values and are strategically distributed across the input space. These inputs enable thorough exploration of system behaviour and help identify potential weaknesses.

Keywords

test input generation, surrogate modelling, equivalence class partitioning, boundary value analysis

1. Introduction

Cyber-Physical Systems (CPSs) are the next generation of engineered systems that integrate computational and physical capabilities [2]. They are applied in aerospace, transportation, healthcare, critical infrastructure, and industrial manufacturing [3]. CPS research has attracted growing interest from academia, industry, and government for its promise of social, economic, and environmental benefits. By combining computation, communication, and control, CPSs enable new ways to monitor, adapt, and influence the physical world, driving future technological advances.

A key characteristic of CPSs is the close integration of various technologies, resulting in large-scale, diverse, and highly complex systems. To ensure proper operation, CPSs must be safe, reliable, and behave according to their specified requirements. This makes thorough testing essential before deployment in real-world environments. Testing CPSs presents two main challenges [4]. First, translating user and consistency requirements into concrete test inputs is often difficult. Second, determining whether test outcomes meet expectations can be ambiguous. Both challenges are amplified by the vast input space and complex correctness criteria typical of CPSs. Consequently, testing methods must accurately capture continuous system dynamics and the ongoing interactions between the system and its environment.

Model-based testing offers a structured approach to address these challenges. This approach uses models to automatically generate test inputs and define conformance criteria for assessing system behaviour during or after test execution. However, the development of models for CPSs remains challenging due to their diversity and complexity. Traditional physics-based modelling approaches require significant manual effort, domain expertise, and computational skills, all of which increase with system complexity [1]. In response, data-driven modelling has emerged as a promising alternative. This approach enables automatic model generation from system data, reducing reliance on manual work and specialised knowledge.

linh.le@tuhh.de (L. N. Le); maximilian.schmidt@tuhh.de (M. Schmidt); goerschwin.fey@tuhh.de (G. Fey)
https://github.com/linhngocle2000 (L. N. Le)
0009-0005-4532-7669 (M. Schmidt); 0000-0001-6433-6265 (G. Fey)

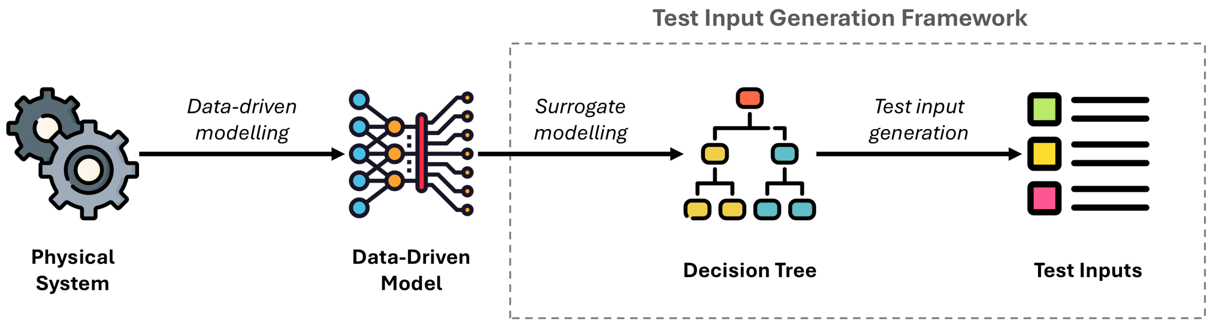


Figure 1: Abstract workflow of our test input generation framework.

This paper presents a test input generation framework based on data-driven models for CPSs. The framework utilises decision trees as white-box models to guide the generation of test inputs. Specifically, it partitions the input domain of the System Under Test (SUT) into equivalence classes, based on the internal structure of the decision tree. Each class represents a group of inputs that follow the same decision path in the tree. Test inputs are sampled statistically from each class, guided by decision tree coverage criteria and the distribution of samples across leaf nodes. Additionally, we propose a method for training a decision tree surrogate from a data-driven model. Surrogate models are simplified analytical representations that approximate the behaviour of complex, computationally intensive models. The resulting surrogate allows application of the same testing strategy across various types of data-driven models.

Figure 1 presents a high-level overview of the test input generation process within our framework. The framework produces test inputs based on the data-driven model of a CPS. This model is derived by applying a data-driven modelling process to the physical system. Once obtained, the model is fed into the framework, which employs a surrogate modelling approach to construct a decision tree. Subsequently, our test input generation method is applied to this decision tree to produce the final test inputs.

We compare the test inputs generated by our framework with those produced by four baseline methods, each employing different random sampling strategies. The evaluation considers three criteria: the effectiveness of the test inputs, the extent of input space coverage, and the distribution of inputs within the covered space. Simulation results indicate that, in the majority of test executions, our framework outperforms the baselines in generating challenging inputs that effectively push the SUT towards its operational limits. These results further demonstrate that the framework enables thorough exploration of the input space by generating inputs that span a broad range of values and are strategically dispersed according to user-defined criteria.

Contributions. The main contributions of this paper are as follows:

- We present a systematic test input generation approach that exploits decision trees by applying equivalence class partitioning.
- We propose a training mechanism that constructs a decision tree surrogate for any given data-driven model.
- We empirically evaluate our framework, which combines surrogate modelling with test input generation, against baseline methods.

The remainder of this paper is structured as follows. Section 2 defines key terms and provides background on decision trees. Section 3 reviews related work. Section 4 introduces our method for generating test inputs from data-driven models. Section 5 describes the software implementation, including the framework workflow and architecture. Section 6 outlines the simulation setup used to examine our research questions. Section 7 analyses and evaluates the experimental results. Finally, Section 8 concludes the paper.

Our test input generation framework, along with the experimental results, is publicly released under the BSD 3-Clause license on GitHub[1].

2. Background

This section explains the terminology used throughout this paper and reviews decision trees with an emphasis on their structure.

2.1. Terminology

This paper presents a framework for generating test inputs based on a data-driven model of a Cyber-Physical System (CPS), as illustrated in Figure 1. A data-driven model is constructed primarily from data collected from the physical system and is used to represent its behaviour. Our proposed framework employs a surrogate modelling approach to build a decision tree from the data-driven model. The framework then uses the internal structure of this tree to generate test inputs. Surrogate models are simplified analytical representations that approximate the behaviour of complex, computationally intensive systems. Throughout this paper, we refer to the physical system being tested as the *System Under Test* (SUT). The data-driven model representing the SUT is termed the *Model Under Test* (MUT), treated as a black-box abstraction. This distinction separates the MUT from a decision tree model used specifically for test input generation. We also refer to such a decision tree model as the *decision tree surrogate* or *white-box surrogate*, as it approximates the MUT and provides a transparent structure for generating test inputs.

Let the input space of a data-driven model be defined as $\mathcal{X} = X_1 \times X_2 \times \cdots \times X_n$, where each X_i represents the domain of the i-th input feature. A *test input* is a vector

$$\mathbf{x} = (x_1, x_2, \ldots, x_n) \in \mathcal{X}, \tag{1}$$

such that each $x_i \in X_i$ denotes the value assigned to the i-th feature, for $i \in \{1, 2, \ldots, n\}$. Depending on the selected coverage criterion, these inputs may include normal, boundary, or invalid values. In contrast, a *test case* consists of input values, execution conditions, and expected outcomes designed for a specific test objective [5]. In the context of software testing, a *test set* is a collection of individual test inputs used to evaluate a specific feature, function, or behaviour of the SUT. In this paper, we define a test set as

$$T = \{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(m)}\} \subseteq \mathcal{X}, \tag{2}$$

where each $\mathbf{x}^{(i)} \in \mathcal{X}$ is a test input, for $i \in \{1, 2, \ldots, m\}$. The test set T must satisfy a specified decision tree coverage criterion C, such that $C(T) = \texttt{true}$. Finally, *test input generation* refers to the process of creating test inputs that effectively verify the behaviour of the SUT.

2.2. Decision Tree

Decision trees are widely used non-parametric models in machine learning, particularly for classification and regression tasks. In classification, the goal is to assign a discrete label to an input instance, whereas regression involves predicting a continuous numerical value, often referred to as the target [6]. When applied to classification problems, the model is known as a *classification tree*; for regression tasks, it is called a *regression tree.*

Definition 2.1 (Decision Tree [7])**.** A decision tree is a tree $T = (V, E)$, where V is the set of nodes and $E \subseteq V \times V$ is the set of directed edges connecting them. The tree represents a predictor

$$d : \mathcal{X} \to \mathcal{Y},$$

[1]TODO

which maps a feature vector $\mathbf{x} \in \mathcal{X}$ to an output label $y \in \mathcal{Y}$. Each feature vector is defined as

$$\mathbf{x} = (x^{(1)}, x^{(2)}, \dots, x^{(D)}),$$

where D is the number of measurable attributes. For classification tasks, $y \in C$, where $C = \{c_1, c_2, \dots, c_n\}$ is a finite set of n class labels; for regression tasks, $y \in \mathbb{R}$.

A decision tree learner constructs the tree using a training dataset of labeled examples, denoted as $\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$. The construction process involves recursively partitioning the dataset in distinct subsets based on some split values in the features. At each step, the learner evaluates all available features and their potential split values. The algorithm selects the features and split values that optimise a predefined heuristic measure. This process allows the tree to divide the input space into regions that correspond to different output labels.

A decision tree consists of three main structural components: nodes, branches, and paths [8]. *Nodes* represent subsets of the training data and are categorised into two types: leaf nodes and inner nodes. Leaf nodes are defined as

$$V_L = \{v \in V \mid \nexists\, (v, v') \in E\}, \tag{3}$$

i.e., nodes with no outgoing edges. Each leaf node $v \in V_L$ is assigned a label $y_v \in \mathcal{Y}$ based on the training data it contains and represents the final prediction after a sequence of decisions. The set of inner nodes is given by

$$V_I = V \setminus V_L. \tag{4}$$

Each inner node $v \in V_I$ applies a split condition which partitions its data subset into disjoint child subsets based on feature values. The topmost inner node $v_{\text{root}} \in V_I$ is called the root node and represents the entire training dataset. *Branches* correspond to edges $(v, v') \in E$. Each branch encodes one outcome of a split condition and directs the flow from a parent node $v \in V$ to a child node $v' \in V$. *Paths* are sequences

$$\pi = (v_0, e_1, v_1, \dots, e_k, v_k), \tag{5}$$

where $v_0 = v_{\text{root}}$, each edge $e_i = (v_{i-1}, v_i) \in E$ for $i \in \{1, \dots, k\}$, each intermediate node $v_j \in V_I \setminus \{v_{\text{root}}\}$ for $j \in \{1, \dots, k-1\}$, and the final node $v_k \in V_L$. Each path encodes a decision rule that determines the prediction for a given input instance.

To make a prediction, the model starts at the root and follows the path determined by evaluating the split conditions at each inner node. Once a leaf is reached, the label associated with that leaf is returned.

In our approach, we employ binary decision trees to guide targeted test input generation. In a binary decision tree, each inner node branches into exactly two child nodes. This binary structure simplifies the decision-making process by evaluating a single condition at each node, such as whether a feature value is less than or greater than a specified threshold. Throughout this paper, we refer to the feature used for splitting at an inner node as the *split feature*, and the corresponding threshold as the *split threshold.* Each split condition is defined by a pair consisting of a split feature and a split threshold, which collectively determines the direction of traversal in the tree.

Hoeffding Tree. In this paper, we propose a method to train a decision tree surrogate from the MUT based on Hoeffding trees. This method is integrated into our test input generation framework, making it applicable to any data-driven models.

Hoeffding trees are an incremental decision tree learning algorithm designed specifically for mining data streams [9]. Unlike traditional decision tree learning methods, which require access to the entire dataset to construct a tree, Hoeffding trees operate in an online fashion. They update the tree incrementally using individual data instances as they arrive, without revisiting previously seen instances. This property makes them particularly suitable for environments where data is generated continuously and potentially without bound.

The key challenge in learning from data streams is making accurate split decisions at each node based on limited observations. Hoeffding trees address this challenge using a statistical tool known as the

Hoeffding bound. The Hoeffding bound provides a probabilistic guarantee that, with high probability, the feature selected for splitting at a node based on limited examples is the same as the one chosen using infinite examples. For a detailed explanation of the Hoeffding bound and its role in Hoeffding trees, see [9].

A notable property of Hoeffding trees is their ability to produce decision trees that are asymptotically arbitrarily close to those generated by batch learners (i.e., learners that train on the entire dataset at once). The probability that a Hoeffding tree and a conventional batch learner select different attributes at a node decreases exponentially with the number of examples. This implies that, given sufficient data, the structure of a Hoeffding tree converges to that of a tree trained on the complete dataset.

3. Related Work

This section reviews existing research across two key domains relevant to our work: model-based testing of CPSs and validation methodologies for machine learning models (MLMs).

Model-based Testing of CPSs. Model-based testing approaches are commonly used to generate test cases from models that describe the architecture, design, or behaviour of a system, based on its specification [5]. Deshmukh et al. [10] propose a Bayesian optimisation framework that constructs a surrogate model and uses an acquisition function to select test inputs. Menghi et al. [11] integrate falsification techniques with surrogate modelling to identify inputs that violate system requirements. Kosek and Gehrke [12] develop an anomaly detection method using artificial neural networks to identify rare data instances that deviate from expected patterns. Their neural network is trained on non-faulty data, resulting in a model that accurately approximates the behaviour of a correct system. Carter et al. [13] present an automated testing framework that generates test cases using statistical models. These models simulate expected user-system interactions based on targeted usage scenarios. Araujo et al. [14] propose an approach that employs hybrid models to identify inputs which maximise the distance between test outputs and expected outputs. Such inputs are likely to drive the system into challenging states, increasing the chance of error detection. The hybrid models represent the desired behaviour of CPSs at a higher level of abstraction, making input generation feasible. Zhang et al. [15] apply formal methods to generate test cases for CPS conformance testing. Their approach uses differential dynamic logic to specify system behaviour and constructs a formal model from these specifications to derive test cases.

Although our proposed approach also relies on models to generate test inputs, the nature of these models differs from those used in [13, 14, 15]. Our approach uses a model that approximates the behaviour of a physical system based on observed data. As such, it cannot reliably produce an expected system response for a given test input. Consequently, our method generates test inputs only, rather than complete test cases that include expected outcomes. In contrast, the models in [13, 14, 15] represent the desired behaviour of a system, as defined by the system specification. By producing expected outcomes for given test inputs, these models support the generation of complete test cases. However, developing such models for CPSs is often challenging, as the process demands extensive domain expertise and is difficult to automate.

Our approach is more closely aligned with those proposed in [10, 11, 12]. Similar to these studies, we build surrogate models from data-driven models to generate test inputs. However, these approaches use surrogate models to approximate system outputs and apply additional techniques to identify promising test inputs based on input-output pairs. By comparison, our method directly leverages the internal structure of the surrogate model to guide test input generation.

Cukic et al. [16] propose a method that automatically generates new test data from an existing set of test data using regression models. The new test data is statistically similar to the data in the original set, but sufficiently different to represent additional test cases. This represents a novel model-based testing approach compared to conventional methods used for CPSs. Rather than modelling system behaviour, as in our approach, their method builds a model from historical test data to generate new test data.

A recent survey [5] by Sadri-Moshkenani et al. provides a comprehensive overview of model-based test generation techniques for CPSs. The survey highlights the methodological characteristics of existing approaches and outlines open research challenges in scalability, coverage, and automation.

Validation Methodologies for MLMs. Recent studies explore property-driven validation techniques for machine learning models by employing surrogate models to guide test input generation. The work presented by Durelli et al. [17] is closely aligned with our research and shares several key similarities with the methodology described in this paper. In particular, Durelli et al. propose a method that constructs a decision tree from the training data of an MLM and encodes tree traversal rules as logical formulae. These formulae are transformed into properties, which are used to systematically generate test suites. Analogous to our approach, their method derives test properties from the internal structure of a white-box surrogate model, eliminating the need for manual property specification. However, unlike our approach, their method does not leverage the statistics available at each node of the decision tree. As a result, test inputs are uniformly allocated across properties, without considering the varying significance of each property. Additionally, their approach only applies to classification models and cannot handle the continuous outputs typical of CPSs.

Sharma et al. [18] present MLCheck, a property-driven testing framework that includes a domain-specific language for property specification and a systematic test generation mechanism. Similar to our approach, MLCheck treats the MUT as a black-box and trains a white-box model (either a decision tree or a neural network) approximating the MUT by using its predictions. The white-box model in MLCheck is used to verify properties and extract counterexamples as test inputs for the original model. While effective, MLCheck requires testers to manually specify the properties under interest, which demands extensive domain knowledge. Furthermore, the framework focuses on property verification rather than using the surrogate model directly for test input generation, as is done in our approach.

Aggarwal et al. [19] propose a related approach that also involves constructing a decision-tree surrogate from the MUT. However, unlike our approach, their methodology applies dynamic symbolic execution to the tree to automatically generate test inputs.

4. The Methodology

In this section, we first present a method for generating test inputs based on data-driven models by exploiting the internal structure of decision trees. To illustrate the approach, we provide a concrete example. Finally, we describe the procedure for training a decision tree surrogate from any given data-driven model. This allows our test input generation technique to be applied broadly across different model types.

Decision trees are employed as white-box surrogate models to guide the test input generation process due to their high interpretability. A surrogate model is a simplified analytical representation that approximates the behaviour of a complex, computationally intensive model. Interpretability, in this context, refers to the degree to which a human can understand the decision-making process of a machine learning model [20]. Decision trees offer a high degree of interpretability through their hierarchical structure, which partitions the input space using simple, transparent decision rules. This transparency facilitates reasoning about the behaviour of the black-box MUT, thereby enabling the identification of informative test inputs.

The partitioning of input space by decision trees closely resembles the principle of equivalence class partitioning. Equivalence class partitioning is a testing technique that divides the input domain into a finite set of classes [21]. Each class represents a group of inputs that are processed similarly by the system. Instead of testing all possible inputs, representative values from each equivalence class are selected. This reduces the number of test cases significantly while maintaining broad coverage across input scenarios. This strategy is particularly effective for testing CPSs, which often exhibit large, continuous input spaces. By exploiting the structural properties of decision trees, we can infer equivalence classes over the input domain and systematically generate test inputs that are both diverse and representative.

4.1. Test Input Generation

Our test input generation method consists of two main phases. The first phase involves extracting equivalence classes from a decision tree. The second phase samples test inputs based on these classes and a specified coverage criterion.

Algorithm 1: Extraction of Equivalence Classes from a Decision Tree

```
1  Function ExtractEquivalenceClasses($T$, specs):
       Input  : Decision tree $T$; system specifications specs
       Output : Set of equivalence classes $C$
2      $P \leftarrow$ CollectAllPaths($T$.root) ;            // Extract all root-to-leaf paths
3      $C \leftarrow \emptyset$ ;                                   // Initialise class set
4      foreach path $\in P$ do
5          $eq\_class \leftarrow$ InitEquivalenceClass(specs);
6          $parent \leftarrow$ path(0) ;                             // Start at root
7          $m \leftarrow$ length(path);
8          for $i \leftarrow 1$ to $m$ do
9              $node \leftarrow$ path($i$) ;
10             UpdateEquivalenceClass($eq\_class, parent, node$) ;
11             $parent \leftarrow node$ ;
12         end
13         $C \leftarrow C \cup \{eq\_class\}$;
14     end
15     return $C$

16 Procedure InitEquivalenceClass(specs):
       Input  : System specifications specs
       Output : Initialised equivalence class $eq\_class$
17     $F \leftarrow$ specs(metadata).features ;                // Extract feature list
18     foreach $feature \in F$ do
19         $eq\_class(feature) \leftarrow$ new FeatureInterval();
20         $eq\_class(feature)$.max $\leftarrow$ specs(feature).max;
21         $eq\_class(feature)$.max_modified $\leftarrow$ false;
22         $eq\_class(feature)$.min $\leftarrow$ specs(feature).min;
23         $eq\_class(feature)$.min_modified $\leftarrow$ false;
24     end
25     return $eq\_class$

26 Procedure UpdateEquivalenceClass($eq\_class, parent, node$):
       Input  : Equivalence class $eq\_class$; parent node $parent$; current node $node$
27     if $parent$.child_left $= node$ then
28         $eq\_class(parent$.feature).max $\leftarrow parent$.threshold;
29         $eq\_class(parent$.feature).max_modified $\leftarrow$ true;
30     else
31         $eq\_class(parent$.feature).min $\leftarrow parent$.threshold;
32         $eq\_class(parent$.feature).min_modified $\leftarrow$ true;
33     end
```

The process of extracting equivalence classes is described in Algorithm 1, Lines 1–15. The algorithm first identifies all possible paths in the decision tree. Each path corresponds to a distinct equivalence

Algorithm 2: Collection of All Root-to-Leaf Paths in a Decision Tree [22]

```
1  Function CollectAllPaths(root):
       Input  : Root node of a decision tree root
       Output : Set of all root-to-leaf paths $P$
2      $P \leftarrow \emptyset$ ;                                   // Initialise path set
3      $path \leftarrow \emptyset$ ;                               // Temporary path container
4      CollectPaths(root, $path, P$) ;                          // Recursive traversal
5      return $P$

6  Procedure CollectPaths(node, $path, P$):
       Input  : Current node node; current path $path$; path set $P$
7      $path \leftarrow path \cup$ node ;                          // Append current node
8      if node.child_left = None $\wedge$ node.child_right = None then
9          $P \leftarrow P \cup \{path\}$ ;                  // Leaf node reached, store path
10     else
11         CollectPaths(node.child_left, $path, P$) ;           // Explore left subtree
12         CollectPaths(node.child_right, $path, P$) ;         // Explore right subtree
13     end
14     $path$.pop() ;                                       // Backtrack for recursion
```

class. For each path, the algorithm initialises a new class instance and traverses the associated nodes, updating the class intervals based on split thresholds and the direction of traversal.

The initialisation of an equivalence class instance is detailed in Algorithm 1, Lines 16–25. The procedure begins by extracting input features from the system specification. For each feature, the class assigns value ranges based on the limits defined in the specification. Flags are also set to indicate that these ranges originate from the specification rather than the decision tree.

The procedure for updating an equivalence class instance is presented in Algorithm 1, Lines 26–33. The procedure first determines whether the current node is the left or right child of its parent. This distinction reveals how the node satisfies the split condition at the parent node. Based on this, the procedure adjusts either the upper or lower bound of the relevant value range. The corresponding flag is set to indicate that the updated bound is derived from the decision tree. The split feature at the parent node determines which value range is affected during this update.

Algorithm 2 [22] illustrates how our method identifies all paths in a decision tree. The algorithm performs a recursive traversal of the decision tree, starting at the root and appending each visited node to the current path. Once a leaf node is reached, the complete path is stored. After exploring both the left and right subtrees, the function backtracks to search for alternative paths.

Decision Tree Coverage Criteria. Once equivalence classes are extracted from the decision tree, our method samples test inputs from them. To ensure the generation of effective inputs, we present two coverage criteria [17]. The first is *Decision Tree Coverage*, defined as follows:

Definition 4.1 (Decision Tree Coverage (DTC))**.** A test set is DTC-adequate if the execution of its inputs causes the model to traverse every path from the root to each leaf node at least once.

In essence, a DTC-adequate test set ensures that the decision tree predicts every possible label at least once. This guarantees that all equivalence classes are covered during testing.

The second criterion is *Boundary Value Analysis*, defined as follows:

Definition 4.2 (Boundary Value Analysis (BVA))**.** A test set is BVA-adequate if its inputs are sampled from the boundaries of all equivalence classes, specifically those boundaries derived from the structure of the decision tree.

BVA complements equivalence class partitioning by targeting the edges of input domains. In this context, boundaries derived from the tree structure correspond to the split conditions of the decision tree. These split features and thresholds are considered informative for predicting the output label of an input instance. Consequently, even small perturbations in input values near these boundaries can lead to different outputs from the decision tree. Such boundary regions in the input space are often sensitive and may reveal weaknesses in the behaviour of the system. By testing inputs in these regions, BVA can help identify potential vulnerabilities in the SUT.

Algorithm 3: Sample Generation from an Equivalence Class Guided by Decision Tree Coverage

```
Function GenerateDTCSamples($eq\_class, n$):
    Input  : Equivalence class $eq\_class$; number of test inputs $n$
    Output : Set of sample vectors $I$
    $I \leftarrow \emptyset$ ;                    // Initialise set of sample vectors
    for each $feature\_range \in eq\_class$ do
        $min \leftarrow feature\_range$.min;
        $max \leftarrow feature\_range$.max;
        if $feature\_range$.min_modified = true then
            $samples \leftarrow$ GenerateRandomSamples(($min$, $max$], $n$);
            $I \leftarrow I \cup \{samples\}$;
        else
            $samples \leftarrow$ GenerateRandomSamples([$min$, $max$], $n$);
            $I \leftarrow I \cup \{samples\}$;
        end
    end
    return $I$
```

Following the extraction of equivalence classes (Algorithm 1), the second phase of our test input generation method focuses on sampling test inputs based on a specified coverage criterion. This phase utilises the equivalence classes derived from decision tree paths to guide the sampling process.

Algorithms 3 and 4 describe two sampling strategies: Decision Tree Coverage (DTC) and Boundary Value Analysis (BVA), respectively. Both algorithms produce a set of sample vectors for a given equivalence class. Each vector contains values sampled according to the feature-specific value range defined by the class. These vectors are then combined to form complete test inputs by aligning values at corresponding positions across all vectors.

We first describe the DTC-based sampling procedure. The algorithm samples input values from a given equivalence class, assuming that each decision tree node applies a split condition of the form `split_feature <= split_threshold`. For each feature in the class, the algorithm examines its value range and determines whether the lower bound is derived from the tree structure. If so, values are randomly sampled from the left-open interval $(min, max]$, where min and max denote the lower and upper bounds of the value range. Otherwise, sampling is performed over the closed interval $[min, max]$. These intervals are designed to ensure that the resulting test inputs follow the decision tree path corresponding to the equivalence class, as defined by DTC. Each set of sampled values is stored in a sample vector and added to the set of sample vectors.

Next, we describe the BVA-based sampling procedure. The algorithm samples input values around the boundaries of a given equivalence class. For each feature in the class, the algorithm checks whether the bounds are derived from the decision tree. If both bounds are derived from the tree, values are sampled around both the upper and lower boundaries. If only one bound is derived, sampling occurs around that boundary. This condition ensures that values are distributed around decision boundaries, consistent with BVA principles. If neither bound originates from the tree, values are sampled from the global feature range defined in the specification. Although these bounds are not critical for boundary

Algorithm 4: Sample Generation from an Equivalence Class Guided by Boundary Value Analysis

```
1 Function GenerateBVASamples($eq\_class, \epsilon, n$):
      Input  : Equivalence class $eq\_class$; boundary neighbourhood size $\epsilon$; number of test inputs $n$
      Output : Set of sample vectors $I$
2     $I \leftarrow \emptyset$ ;                                  // Initialise set of sample vectors
3     for each $feature\_range \in eq\_class$ do
4         $min \leftarrow feature\_range$.min;
5         $max \leftarrow feature\_range$.max;
6         if $feature\_range$.max_modified = true and $feature\_range$.min_modified = true then
7             $k_1 \leftarrow \lceil \frac{n}{2} \rceil$;
8             $k_2 \leftarrow \lfloor \frac{n}{2} \rfloor$;
9             $samples_1 \leftarrow$ GenerateRandomSamples($[max - \epsilon,\ max + \epsilon]$, $k_1$);
10            $samples_2 \leftarrow$ GenerateRandomSamples($[min - \epsilon,\ min + \epsilon]$, $k_2$);
11            $I \leftarrow I \cup \{samples_1 \cup samples_2\}$;
12        else if $feature\_range$.max_modified = true then
13            $samples \leftarrow$ GenerateRandomSamples($[max - \epsilon,\ max + \epsilon]$, $n$);
14            $I \leftarrow I \cup \{samples\}$;
15        else if $feature\_range$.min_modified = true then
16            $samples \leftarrow$ GenerateRandomSamples($[min - \epsilon,\ min + \epsilon]$, $n$);
17            $I \leftarrow I \cup \{samples\}$;
18        else
19            $samples \leftarrow$ GenerateRandomSamples($[min,\ max]$, $n$);
20            $I \leftarrow I \cup \{samples\}$;
21        end
22    end
23    return $I$
```

Algorithm 5: Generate Test Inputs from an Equivalence Class

```
1 Function GenerateTestInputs($I$):
      Input  : Set of $n$ sample vectors $I = \{I_1, I_2, \ldots, I_n\}$, where each $I_j$ contains $m$ samples of feature $j$
      Output : Set of $m$ test inputs $T = \{t_1, t_2, \ldots, t_m\}$, where each $t_i$ contains $n$ feature values
2     $T \leftarrow \emptyset$ ;                                  // Initialise set of test inputs
3     $m \leftarrow$ length($I_1$) ;                   // All sample vectors have equal length
4     for $i \leftarrow 1$ to $m$ do
5         $t \leftarrow \emptyset$ ;                              // Initialise a test input
6         for $j \leftarrow 1$ to $n$ do
7             $t \leftarrow t \cup \{I_j(i)\}$ ;      // Select $i$-th sample from $j$-th sample vector
8         $T \leftarrow T \cup \{t\}$;
9     return $S$
```

testing, sampling is still necessary to ensure that each test input includes a value for every feature. Each set of sampled values is stored in a sample vector and added to the set of sample vectors.

The set of sample vectors generated by Algorithms 3 and 4 serve as input to Algorithm 5, which constructs complete test inputs. Each test input is formed by aggregating values at the same index across all sample vectors, resulting in a tuple that includes one value per feature. To generate a DTC-adequate or BVA-adequate test set, the sampling process (Algorithm 3 or 4) and the combination process

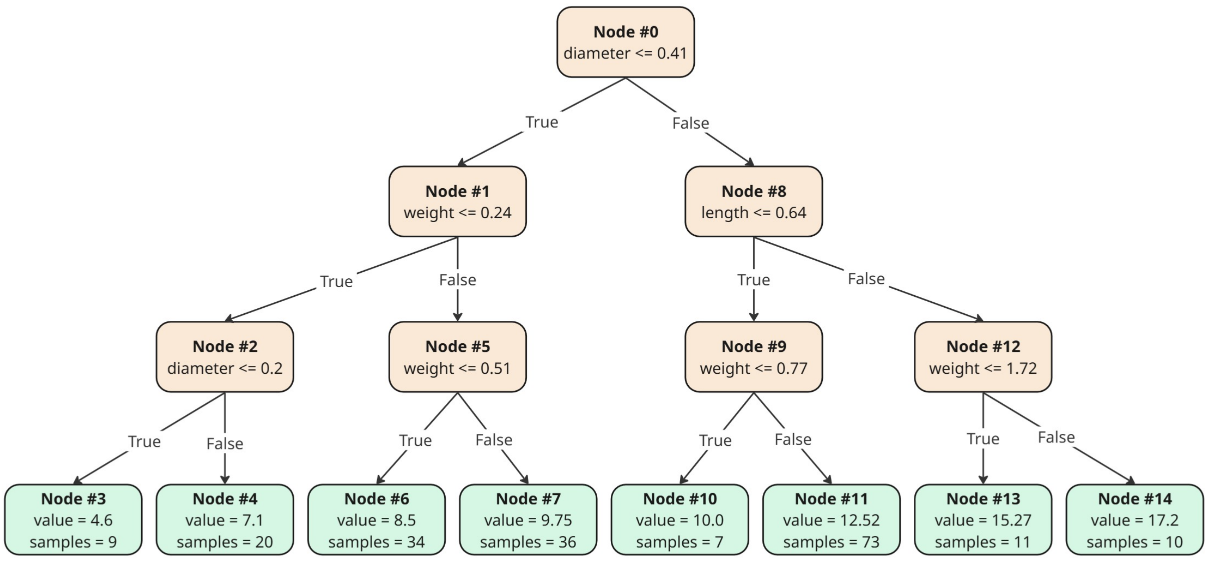


Figure 2: Decision tree generated from the Abalone dataset. Leaf nodes are highlighted in green and inner nodes in orange. Node #0 represents the root. Each leaf node displays the predicted age of the abalone and the number of training samples that reached it during model training.

(Algorithm 5) are repeated for each equivalence class extracted from the decision tree.

Our method allocates a variable number of test inputs to each equivalence class. The allocation is based on the number of training examples that reach the leaf node corresponding to the path from which the equivalence class is derived. We define an equivalence class as dominant if its corresponding leaf node contains more training examples than others. Dominant classes represent regions of the input space that are frequently accessed during system operation. Therefore, it is important to sample more test inputs from these regions to ensure reliable performance under typical conditions. Conversely, equivalence classes associated with leaf nodes containing fewer training examples represent less commonly accessed regions of the input space. Sampling more test inputs from these regions can help reveal hidden errors and improve system robustness under edge-case conditions. To support both testing objectives, we propose two allocation strategies. The tester may select the appropriate strategy based on the test requirements and objectives.

In the first strategy, dominant equivalence classes receive more test inputs:

Definition 4.3 (Proportional Allocation Function)**.** Let $S = \{s_1, s_2, \ldots, s_m\}$ be a set of integers, where each s_j denotes the number of training examples reaching the leaf node of path j. Let $[x]$ denote the Banker's rounding function [23], which rounds x to the nearest integer; if x is exactly halfway between two integers, it is rounded to the nearest even integer. Let t be the total number of test inputs to be generated. The number of test inputs n_j allocated to equivalence class j is computed as:

$$n_j = \left[t \cdot \frac{s_j}{\sum_{i=1}^{m} s_i} \right] \qquad \forall\, 1 \leq j \leq m.$$

In the second strategy, less dominant equivalence classes receive more test inputs:

Definition 4.4 (Inverse Allocation Function)**.** Let $N = \{n_1, n_2, \ldots, n_l\}$ be a sorted set of integers such that $n_1 \leq n_2 \leq \ldots \leq n_l$, where each n_j denotes the number of test inputs allocated to equivalence class j, as computed using Equation (**??**). The number of test inputs m_j assigned to equivalence class j is defined as:

$$m_j = n_{l-(j-1)}.$$

Table 1
Equivalence classes extracted from the Abalone decision tree. The final two columns indicate the number of test inputs to be generated for each class, calculated according to Definitions 4.3 and 4.4, respectively. The total number of test inputs is set to $t = 100$. The other columns specify the upper and lower bounds of the value ranges for each feature within each equivalence class. Bounds highlighted in bold are externally specified by the tester.

Class	Length		Diameter		Weight		# Test Inputs (Def. 4.3)	# Test Inputs (Def. 4.4)
	min	max	min	max	min	max		
1	**0.17**	**0.73**	**0.13**	0.2	**0.03**	0.24	4	36
2	**0.17**	**0.73**	0.2	0.41	**0.03**	0.24	10	6
3	**0.17**	**0.73**	**0.13**	0.41	0.24	0.51	17	5
4	**0.17**	**0.73**	**0.13**	0.41	0.51	**2.55**	18	4
5	**0.17**	0.64	0.41	**0.58**	**0.03**	0.77	4	18
6	**0.17**	0.64	0.41	**0.58**	0.77	**2.55**	36	4
7	0.64	**0.73**	0.41	**0.58**	**0.03**	1.72	6	10
8	0.64	**0.73**	0.41	**0.58**	1.72	**2.55**	5	17

4.2. A Motivating Example

We demonstrate our test input generation approach using a decision tree constructed from the Abalone dataset [24]. Instead of training a model on system-specific input and output data, we use the Abalone dataset for its simplicity. This dataset contains physical measurements of abalones, and the task is to predict their age based on these measurements. The relevant features are:

- Length – the longest shell measurement (in mm)
- Diameter – measured perpendicular to the length (in mm)
- Weight – the total weight of the abalone (in grams)

Figure 2 presents the decision tree generated from this dataset. The tree consists of 15 nodes: eight leaf nodes (highlighted in green) and seven inner nodes (highlighted in orange). The topmost inner nodes, labeled as node #0, is the root node. Each inner node defines a split condition of the form `split_feature <= split_threshold`. Input instances that satisfy this condition are directed to the left child node for further evaluation; otherwise, they proceed to the right child node. Each leaf node contains a predicted age value for the abalone, along with a sample count indicating how many training examples reached that node during training.

When applying our proposed method to the Abalone decision tree, the algorithm first traverses the tree to identify all distinct paths. From each path, it extracts an equivalence class, as described in Algorithm 1. Since the tree contains eight distinct paths, the method yields eight equivalence classes. Table 1 presents these extracted equivalence classes. In the table, equivalence classes are listed top-down, corresponding to tree paths ordered from left to right. Each equivalence class consists of three value ranges, one for each feature in the Abalone dataset. For each range, a lower and upper bound is specified, denoted as min and max in the table. These bounds are derived by analyzing the split conditions of internal nodes along each path. Some features may not appear in any split condition for a given path. For example, the feature Length is not used in the leftmost path. In other cases, only one bound (either upper or lower) can be determined. For instance, in the leftmost path, only upper bounds for Diameter and Weight can be extracted. When a bound cannot be inferred from the tree traversal, it is taken from the specification provided by the tester. These bounds are highlighted in bold in the table. The final two columns in Table 1 show the number of test inputs to be generated for each equivalence class, computed using two different approaches outlined in Definitions 4.3 and 4.4, respectively. For this computation, we set the total number of test inputs to be generated as $t = 100$.

The next phase of our method involves sampling test inputs from the equivalence classes based on a specified coverage criterion. For clarity, we illustrate this process using only Equivalence Class 1, as shown in Table 1. Table 2 lists the feature intervals used for sampling test inputs, as described in

Table 2
Feature intervals used for sampling test inputs generated under Decision Tree Coverage (DTC) and Boundary Value Analysis (BVA). For BVA, a neighbourhood size of $\epsilon = 0.05$ is applied to each boundary value.

Coverage Criterion	Length	Diameter	Weight
DTC	$[0.17, 0.73]$	$[0.13, 0.20]$	$[0.03, 0.24]$
BVA	$[0.17, 0.73]$	$[0.15, 0.25]$	$[0.19, 0.29]$

Algorithms 3 and 4. The intervals vary depending on the chosen coverage criterion: Decision Tree Coverage (DTC) or Boundary Value Analysis (BVA). Intervals generated under DTC are designed to ensure that the sampled values, when used as test inputs, follow the tree path associated with Equivalence Class 1. In contrast, intervals derived from BVA target values within an ϵ-neighbourhood of all boundary values extracted from the same path. For this example, we set $\epsilon = 0.05$. For features whose boundary values are externally specified by the tester, the corresponding intervals follow the DTC-based structure. These boundaries are considered less critical for testing, as they do not appear along the relevant tree path. In Table 2, this applies to the feature Length.

To generate four test inputs for Equivalence Class 1, we sample four values from each DTC interval listed in Table 2. Following the procedure described in Algorithm 3, we construct three sample vectors, each corresponding to a specific feature:

$$\begin{aligned} \text{Length}: &\quad [0.63, 0.40, 0.39, 0.66], \\ \text{Diameter}: &\quad [0.16, 0.17, 0.16, 0.14], \\ \text{Weight}: &\quad [0.08, 0.05, 0.18, 0.20]. \end{aligned}$$

Each vector contains four independently sampled values from its corresponding feature interval. By combining values across these vectors, as described in Algorithm 5, we generate the following test inputs:

$$\begin{aligned} \text{Test Input \#1}: &\ (0.63, 0.16, 0.08), \\ \text{Test Input \#2}: &\ (0.40, 0.17, 0.05), \\ \text{Test Input \#3}: &\ (0.39, 0.16, 0.18), \\ \text{Test Input \#4}: &\ (0.66, 0.14, 0.20). \end{aligned}$$

Each test input is a tuple consisting of values for Length, Diameter, and Weight, respectively. The same procedure can be applied to generate test inputs using intervals derived from BVA, as outlined in Algorithm 4.

4.3. Surrogate Modelling

To ensure our test input generation method is applicable to any data-driven model, we propose a technique for constructing a decision-tree surrogate using incrementally learned Hoeffding Trees. Surrogate models are simplified analytical representations that approximate the behaviour of complex, computationally intensive systems. Our goal is not to replicate the predictions of the black-box MUT with high fidelity. Instead, we aim to optimise split features and thresholds in the surrogate decision tree by analysing the input-output behaviour of the MUT. These splits define input regions represented by equivalence classes, which serve as the basis for generating effective test inputs.

Figure 3 illustrates the workflow for generating a decision-tree surrogate from a data-driven model. The process begins by constructing a dynamic database that pairs original training inputs with outputs produced by the MUT. This database is continuously updated with new data points. Inputs are generated by a data model, and outputs are obtained by querying the MUT. The data model ensures that generated inputs follow the same distribution as the original training data. To build the data model, we use

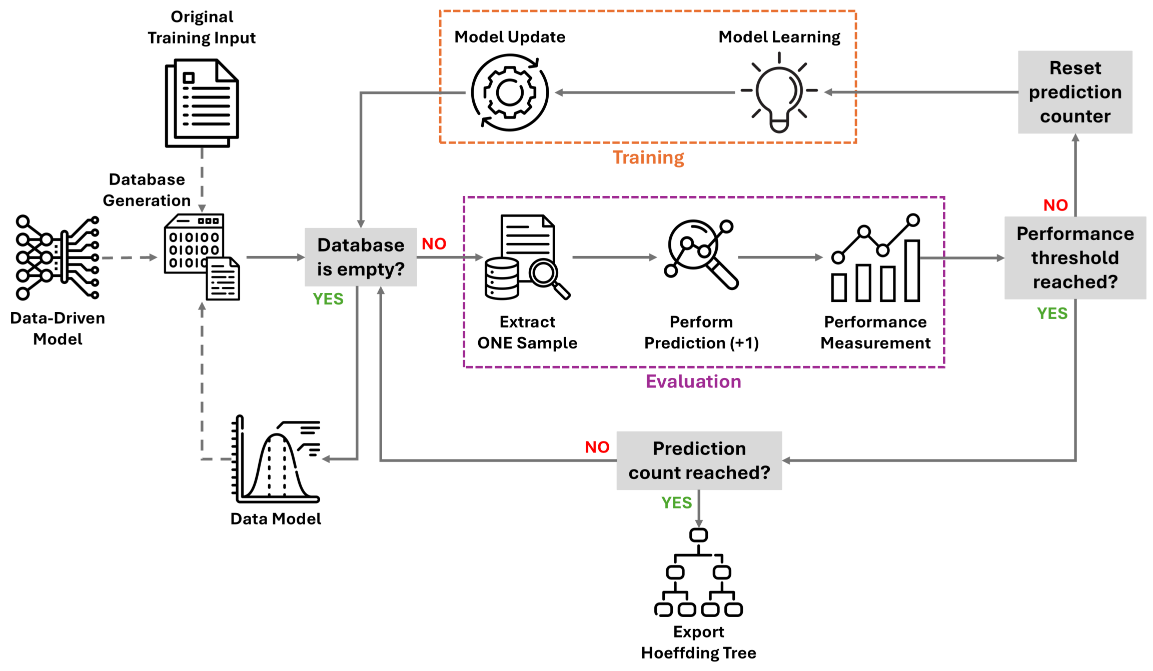


Figure 3: Workflow for constructing a decision-tree surrogate from a data-driven model.

empirical distributions for discrete-valued features and kernel density estimation (KDE) for continuous-valued features [25]. An empirical distribution represents the probability of each value in a dataset. KDE estimates the probability density function (PDF) based on the original training data, using kernels as weighting functions [26]. For our KDE implementation, we use the commonly adopted Gaussian kernel. The estimated PDF is computed as:

$$f(x) = \frac{1}{m} \sum_{j}^{m} \left[\frac{1}{\sqrt{2\pi\sigma}} e^{-\left(\frac{x-\mu_j}{2\sigma}\right)^2} \right], \tag{6}$$

where:

m	is the number of training examples,
μ_j	is the value of the feature for the j-th example,
σ	is the width of the Gaussian kernel.

As the number of training examples increases, the estimated PDF converges to the true distribution. When data is scarce, the method yields near-Gaussian estimates.

Each data point in the database is sequentially passed to the Hoeffding Tree for processing. Initially, the tree is empty, i.e., it contains no nodes or branches, until the first data point triggers its construction. The tree generates a prediction based on the input. This prediction is then compared to the output from the MUT to assess the performance of the current tree. If the performance falls below a predefined threshold, the prediction counter is reset, and the current sample is used to update the tree. If the threshold is met, the algorithm checks whether the required number of correct predictions has been achieved. If so, the Hoeffding Tree is finalised; otherwise, the process continues with new data points. The resulting Hoeffding Tree serves as a white-box surrogate model that approximates the behaviour of the original data-driven model and guides the test input generation process.

This approach offers significant flexibility and control. Due to the incremental nature of Hoeffding Trees, the training and evaluation phases can be managed independently, allowing for precise tuning toward the desired performance. Furthermore, the number of training examples used to construct the surrogate can be adjusted to meet the predefined performance threshold.

Table 3
Parameters of our test input generation framework

Parameter	Type	Explanation
`model`	$M : \vec{X} \to \vec{Y}$	data-driven MUT
`train_set`	$\vec{X}$	training dataset used for MUT
`specification_file`	JSON	input specification file
`coverage_criterion`	{*dtc*, *bva*}	decision tree coverage criterion
`n_test_inputs`	integer	total number of test inputs
`test_allocation`	{*proportional*, *inverse*}	test input allocation strategy
`epsilon`	float	boundary neighbourhood size for BVA

Why Hoeffding Trees? Our surrogate modelling approach involves training a decision tree using examples drawn from an open-ended data stream. This is made possible by models capable of generating an infinite sequence of input-output pairs, which continuously populate the training database. Our goal is to construct a decision tree that meets a predefined performance threshold, based on a specified data-driven model. To achieve this, we require a learning method that operates continuously, examining new examples as they arrive without discarding valuable information. The method must fully utilise all available training data and continue learning until the performance criterion is satisfied. This strategy ensures low computational cost while adapting to data-driven models of varying complexity. For simpler models, fewer training examples may suffice to meet the desired performance, while more complex models typically demand larger datasets. These requirements are well addressed by incremental learning algorithms. Among them, we choose Hoeffding Trees due to their suitability for streaming data and strong theoretical guarantees [9]. As an incremental learner, Hoeffding Trees produce models similar to those obtained through batch learning on the same dataset. Additionally, they are robust to the order of incoming examples, maintaining consistent performance regardless of input sequence. Hoeffding Trees process each example only once and do not require storing data from the stream. They operate in time proportional to the number of features and require only a single pass over the data. As a result, they scale efficiently to extremely large or even infinite datasets with minimal computational overhead. These properties make Hoeffding Trees a well-suited choice for our surrogate modelling method.

5. Implemention

We implement the approach described in Section 4 as a test input generation framework. This section outlines the main parameters of the framework, its workflow, and the underlying software architecture.

The framework is written in Python (v3.10.11) and comprises approximately 1500 lines of code. We use the `river`[2] library (v0.22.0) for incremental training of the decision tree surrogate. For kernel density estimation, used to construct the data model in the surrogate modeling process, we rely on the `scikit-learn`[3] library (v1.6.1).

Table 3 summarises the main parameters of the framework. The core input is a data-driven model representing the SUT, provided via the `model` parameter. This model must implement a `predict()` function that returns an output for a given input instance. This functionality is essential for generating training examples used to train the decision tree surrogate. The training dataset used to train the data-driven model must also be supplied, via the `train_set` parameter. This dataset is required both for constructing the data model and for training the surrogate. The `specification_file` parameter refers to a JSON file that defines the input parameters of the SUT, including metadata such as minimum and maximum values, as well as data types. The `coverage_criterion` parameter specifies the decision tree coverage criterion used for test input generation, as detailed in Section 4.1. The total number of test inputs to be generated is set via `n_test_inputs`. The `test_allocation` parameter defines how

[2]https://riverml.xyz/0.22.0
[3]https://scikit-learn.org/1.6

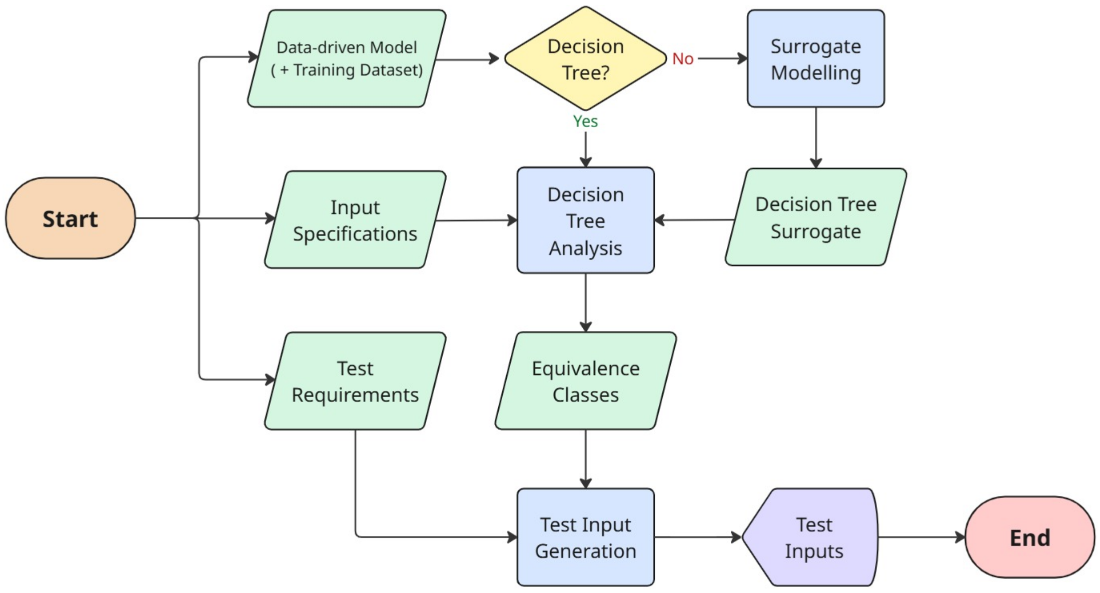


Figure 4: Detailed workflow of our test input generation framework.

test inputs are distributed across equivalence classes, either proportionally or inversely, also described in Section 4.1. Finally, the `epsilon` parameter controls the neighbourhood size around each boundary value used in Boundary Value Analysis (BVA).

5.1. Workflow

Figure 4 provides a detailed overview of the workflow used in our test input generation framework. The process begins with the framework receiving a data-driven model that describes the behaviour of the SUT. If the model is not a decision tree, the tester must also supply the training dataset used to construct the model. In addition, the tester provides a specification file describing the input parameters of the SUT, including their value ranges and data types. The tester must also specify the test requirements by assigning values to the parameters `coverage_criterion`, `n_test_inputs`, `test_allocation`, and `epsilon`, as listed in Table 3. The framework first checks whether the provided model is a decision tree. If not, it applies the surrogate modelling technique described in Section 4.3 to construct a decision tree surrogate. If the model is already a decision tree, it is used directly for test input generation. Next, the framework analyses the internal structure of the decision tree and uses the input specifications to identify equivalence classes, as outlined in Section 4.1. Based on the specified test requirements, the framework applies the method from Section 4.1 to generate test inputs from these equivalence classes. The resulting test inputs are then produced as output.

5.2. Software Architecture

The software architecture of our test input generation framework is structured into multiple layers, each fulfilling a specific role. These layers follow a top-down hierarchy, where higher layers depend on services provided by lower ones, but not the other way around [27]. Modules within a layer may interact with other modules in the same layer or with those below, while remaining independent of any above. This separation of concerns promotes modularity, simplifies development and testing, and improves maintainability.

Figure 5 illustrates the layered architecture of our framework. At the top is the User Interface Layer, responsible for managing user inputs. This layer consists of two modules. The Input Specifications module converts a specification file into a format compatible with the framework. The file contains metadata describing the input parameters of the SUT. Similarly, the Test Requirements module translates the test requirements into a format suitable for further processing. Beneath this lies the Application

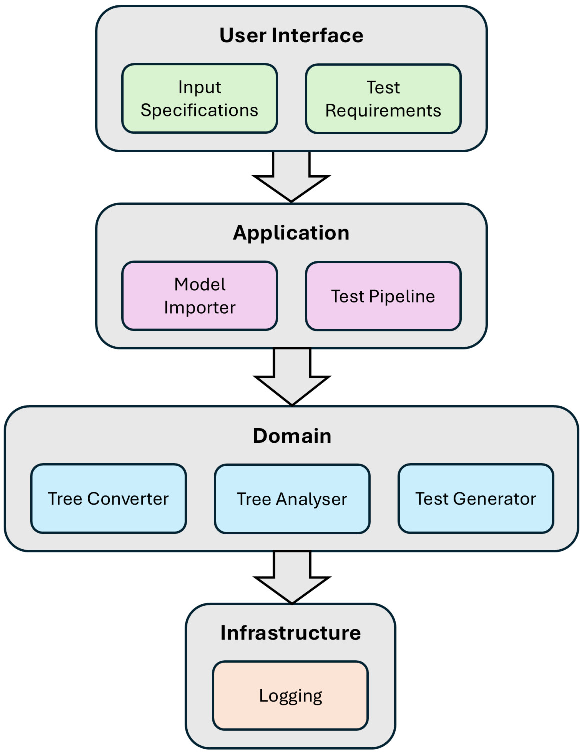
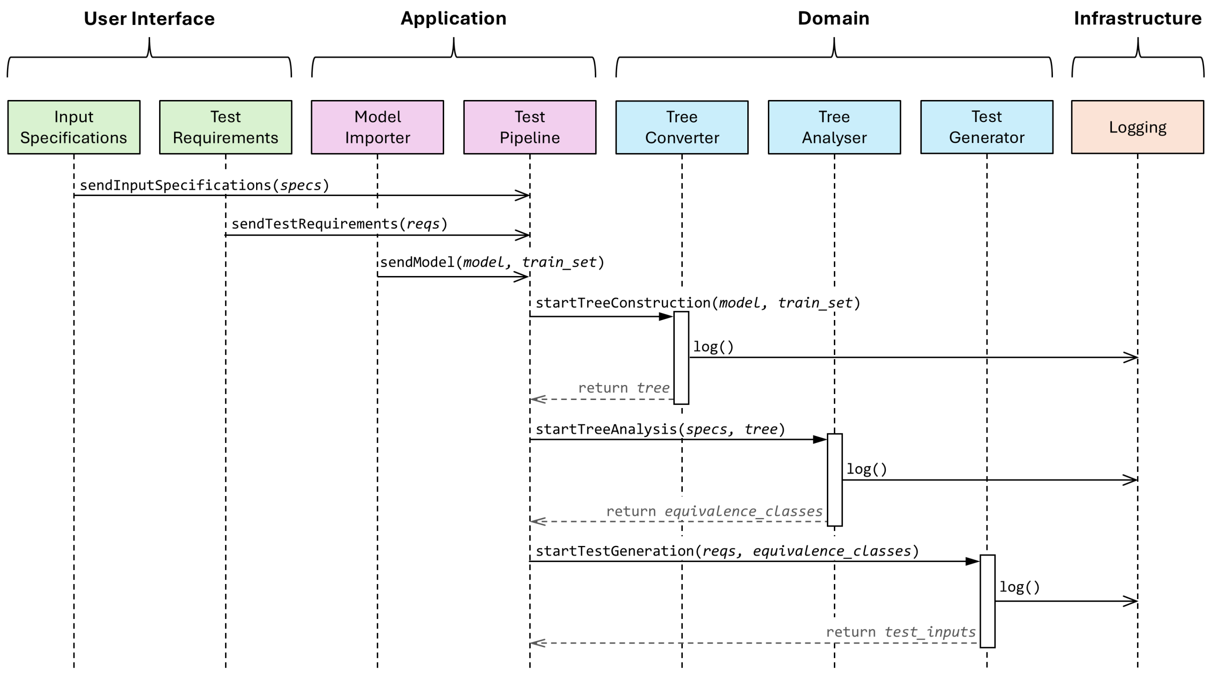


Figure 5: Software architecture of our test input generation framework.

Layer, which integrates the data-driven model and defines the overall workflow. This layer comprises two modules. The Model Importer module provides an interface for importing a data-driven model of the SUT. The module also implements a wrapper function that invokes the `predict()` method of the model. This wrapper function enables modules in the lower layer to easily retrieve model predictions for given input instances. The Test Pipeline module manages execution and assigns relevant tasks to each module within the Domain Layer. The Domain Layer forms the core of the framework and performs its primary functions. This layer consists of three modules. The Tree Converter module applies the surrogate modelling technique described in Section 4.3 to construct a decision tree surrogate when the input model is not already a decision tree. The Tree Analyser module examines the structure of a decision tree to identify equivalence classes. The Test Generator module is responsible for producing test inputs. Finally, the Infrastructure Layer provides technical support for the Domain Layer. The layer includes the Logging module, which records the surrogate model, equivalence classes, and generated test inputs. In addition, the module logs operations throughout execution of the framework for the purpose of monitoring progress and facilitating debugging.

Figure 6 presents a message sequence chart that illustrates interactions between modules across the various layers of the framework. The process begins when the Input Specifications module receives input specifications from the tester and processes them. The formatted data is forwarded to the Test Pipeline module for further handling. Similarly, the Test Requirements module receives and processes the test requirements before passing the formatted output to the Test Pipeline module. Upon receiving a data-driven model of the SUT and its corresponding training data, the Model Importer module transfers both to the Test Pipeline module. The Test Pipeline module is responsible for coordinating the overall workflow of the framework. The module first determines whether the data-driven model is a decision tree. If so, it signals the Tree Converter module to begin the surrogate modelling process, providing the model and training data as inputs. Once the surrogate model has been constructed, the Test Pipeline module initiates the tree analysis phase. The Tree Analyser module receives the decision tree and input specifications, performs structural analysis, and extracts equivalence classes. Upon completion, the Tree Analyser module returns the equivalence classes to the Test Pipeline module. These classes,



User Interface
Application
Domain
Infrastructure
Input Specifications
Test Requirements
Model Importer
Test Pipeline
Tree Converter
Tree Analyser
Test Generator
Logging
sendInputSpecifications(specs)
sendTestRequirements(reqs)
sendModel(model, train_set)
startTreeConstruction(model, train_set)
log()
return tree
startTreeAnalysis(specs, tree)
log()
return equivalence_classes
startTestGeneration(reqs, equivalence_classes)
log()
return test_inputs

Figure 6: Message sequence chart illustrating the interactions between modules across different layers of the framework.

along with the test requirements, are passed to the Test Generator module, which generates the test inputs. The surrogate modelling, tree analysis, and test generation processes are each logged using the infrastructure provided by the Logging module. Once complete, the Test Generator module returns the test inputs to the Test Pipeline module. Finally, the Test Pipeline module delivers the generated test inputs to the tester.

6. Simulation

We aim to evaluate the effectiveness of the test inputs generated by our framework in assessing the performance of the SUT. To achieve this, we compare our test inputs with those produced by four baseline methods. These baselines rely on various random sampling strategies, either from the input space of the SUT or from the training data of a data-driven model. In addition to effectiveness, we assess the input space coverage achieved by our approach. This is done by comparing the distribution of our test inputs with those generated by the baseline methods. To guide our evaluation, we design a series of simulations to address the following Research Questions (RQs):

- **RQ1:** How effective are test inputs generated by our framework without surrogate modelling (i.e., using a decision tree as the input model) compared to those produced by baseline methods?
- **RQ2:** How effective are test inputs generated by our framework with surrogate modelling (i.e., using a decision tree as the input model) compared to those produced by baseline methods?
- **RQ3:** How extensively do our test inputs explore the input space compared to baseline methods, and how are they distributed within that space?

In the following section, we present the hypotheses used to investigate these questions. We then describe the simulation setup used to conduct the experiments. Finally, we review the evaluation metrics used to measure the performance of our framework.

6.1. Hypotheses Formulation

We hypothesise that our framework is capable of generating effective test inputs from data-driven models. This prediction is based on the use of decision tree surrogates within the framework to construct equivalence classes by analysing their internal structure and decision-making information. These classes define subsets of the input space of the SUT, from which representative values are sampled for test input generation. By partitioning the input space, the number of required test inputs is significantly reduced while maintaining comprehensive coverage. Furthermore, the framework applies decision tree coverage criteria and test allocation strategies to guide the sampling process. These mechanisms help generate test inputs that satisfy diverse testing requirements and objectives.

Based on this prediction and the research questions, we formulate the following hypotheses:

H1: Test inputs generated by our framework without surrogate modelling are more effective than those produced by baseline methods.

H2: Test inputs generated by our framework with surrogate modelling are more effective than those produced by baseline methods.

H3: Test inputs generated by our framework achieve greater input space coverage compared to those generated by baseline methods.

H4: Test inputs generated by our framework follow a distribution that reflects the proposed test allocation strategies.

6.2. Evaluation Metrics

To assess the effectiveness of test inputs generated by different methods, we use the F_1 score for classification models and the Mean Absolute Error (MAE) for regression models. Both metrics are widely accepted standards for assessing model performance in their respective domains. In the context of our evaluation, if a model M performs worse on test inputs generated by one approach A_1 than on those produced by another approach A_2, then A_1 is considered more effective than A_2 [17].

Formally, for classification models:

Definition 6.1 (Effectiveness Based on F_1 Score)**.** Given a classification model M and a set of test inputs T, we denote the F_1 score obtained by evaluating M on T as $F_1(M(T))$. Consequently,

$$F_1(M(T_1)) < F_1(M(T_2))$$

indicates that T_1 is more effective than T_2, as it reveals a larger number of misclassifications by M.

For regression models:

Definition 6.2 (Effectiveness Based on MAE)**.** Given a regression model M and a set of test inputs T, we denote the MAE obtained by evaluating M on T as $E(M(T))$. Consequently,

$$E(M(T_1)) > E(M(T_2))$$

indicates that T_1 is more effective than T_2, as it reveals a higher average prediction error in M.

Separate definitions of effectiveness are required for classification and regression models due to the nature of their evaluation metrics. A higher F_1 score reflects better predictive accuracy in classification tasks, whereas a lower MAE indicates higher accuracy in regression tasks.

We evaluate input space coverage by computing the volume of the *Minimum Bounding Box* (MBB) for each set of test inputs. The volume of an MBB provides a measure of how extensively a given set explores the input space. By comparing these volumes across different test input generation strategies, we assess the relative effectiveness of each method in spanning the input space. Formally, the MBB and its volume are defined as follows:

Definition 6.3 (Volume of a Minimum Bounding Box [28])**.** Let $X = \{x_1, x_2, \ldots, x_m\} \subset \mathbb{R}^n$ be a set of m test inputs in an n-dimensional space. Each test input $x_i = (x_{i1}, x_{i2}, \ldots, x_{in})$ represents values across n features. The *Minimum Bounding Box* is the smallest axis-aligned hyperrectangle (i.e., with edges parallel to the coordinate axes) that contains all test inputs in X. Its *volume* V is computed as:

$$V = \prod_{j=1}^{n} \left(\max_{1 \le i \le m} x_{ij} - \min_{1 \le i \le m} x_{ij} \right).$$

Even when the input space is broadly covered, test inputs may still be sparsely distributed, leaving gaps that result in untested regions. To quantify this distribution, we compute the *Coefficient of Variation* (CV) for each set of test inputs and compare the values across different test input generation methods. CV measures the relative variability in a distribution with respect to its mean [29]. A low CV indicates low variability around the mean, suggesting that test inputs are more uniformly spread but may be tightly clustered and concentrated. In contrast, a high CV reflects greater variability, meaning test inputs are unevenly distributed and more widely scattered across different regions of the input space. For this evaluation, the CV is computed from the standard deviation and mean of the pairwise *Mahalanobis distances* between all test inputs in a given set. We formally define the Mahalanobis-based CV as follows:

Definition 6.4 (Mahalanobis-Based Coefficient of Variation)**.** Let $X = \{x_1, x_2, \ldots, x_m\} \subset \mathbb{R}^n$ be a set of m test inputs, where each input $x_i = (x_{i1}, x_{i2}, \ldots, x_{in})$ consists of values across n features. Let $D_M(x_i, x_j)$ denote the Mahalanobis distance between inputs x_i and x_j in the multivariate input space. The mean pairwise Mahalanobis distance is given by:

$$\bar{D} = \frac{2}{m(m-1)} \sum_{1 \le i < j \le m} D_M(x_i, x_j).$$

The corresponding variance is:

$$\sigma^2 = \frac{2}{m(m-1)} \sum_{1 \le i < j \le m} \left(D_M(x_i, x_j) - \bar{D} \right)^2.$$

The standard deviation is then:

$$\sigma = \sqrt{\sigma^2}.$$

Finally, the Coefficient of Variation is defined as:

$$\text{CV} = \frac{\sigma}{\bar{D}}.$$

6.3. Setup

The primary objective of the simulation is to evaluate the effectiveness of the test inputs generated by our framework. To this end, we conduct two distinct simulations: one in which the framework generates test inputs without surrogate modelling, and another in which surrogate modelling is employed prior to test input generation. In addition, we generate test inputs using four baseline methods and compare their effectiveness against those from our framework using a defined set of metrics. We further analyse the coverage and distribution of the test inputs across the input space using a separate set of metrics and provide a comprehensive comparison between our framework and the baseline methods.

Datasets. To construct the data-driven input MUTs used in our framework, we selected 24 diverse datasets, including both classification and regression tasks. These datasets are sourced from the UCI Machine Learning Repository[4] and the online data science platform Kaggle[5]. Table 4 and Table 5 present

[4]https://archive.ics.uci.edu/datasets
[5]https://www.kaggle.com/datasets

Table 4
Overview of the datasets used for classification tasks in the simulation.

Dataset	Features	Instances	Classes
Banknote Authentication	4	1372	2
Breast Cancer Wisconsin	30	569	2
Dry Bean	16	13611	7
Iranian Churn	13	3150	2
Iris	4	150	3
Mammography	6	11183	2
Phishing Websites	30	11055	2
Pima Indians Diabetes	8	2768	2
Seeds	7	210	3
Sonar, Mines vs. Rocks	60	208	2
Vertebral Column	6	310	3
Wireless Localisation	7	2000	4

Table 5
Overview of the datasets used for regression tasks in the simulation.

Dataset	Features	Instances
Appliances Energy	27	19735
Body Fat	14	252
Combined Cycle Power Plant	4	9568
Concrete Strength	8	1030
Forest Fires	29	517
Liver Disorders	5	345
Parkinsons Telemonitoring	19	5875
Real Estate Valuation	6	414
Superconductor Temperature	81	21263
Seoul Bike Sharing Demand	15	8760
Wind Power	8	43800
Wind Speed	7	5638

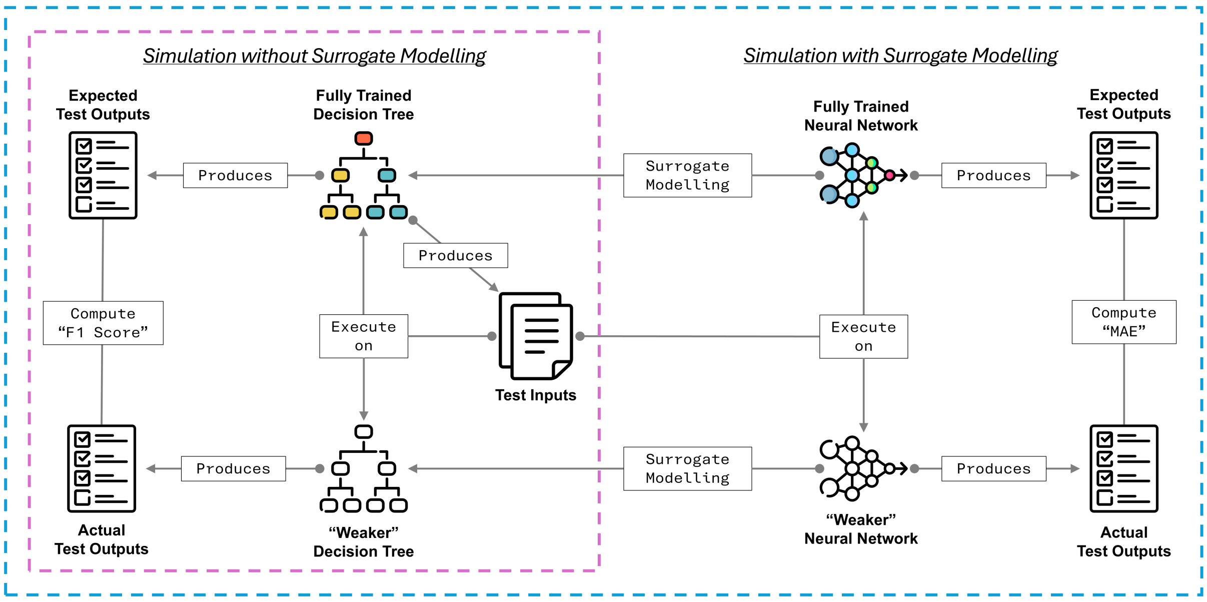


Figure 7: Simulation workflow for evaluating the effectiveness of test inputs generated by our framework on a single dataset.

an overview of the datasets used for classification and regression tasks, respectively, in the simulation. These datasets vary significantly with respect to the number of instances and the number of features. For classification tasks, the number of instances ranges from as few as 150 (Iris) to 13,611 (Dry Bean), indicating a wide variation in dataset scale. The number of features also differ significantly, from 4 (Iris) to 60 (Sonar, Mines vs. Rocks). The number of classes ranges from two in binary classification tasks, such as Banknote Authentication, to seven in multiclass problems like Dry Bean. The regression datasets exhibit similar variability, with instance counts ranging from 252 (Body Fat) to 43,800 (Wind Power), and feature counts from 4 (Combined Cycle Power Plant) to 81 (Superconductor Temperature).

Workflow. To evaluate the effectiveness of the test inputs generated by our framework, we conduct two distinct simulation procedures. Figure 7 illustrates the workflow for both simulations.

In the first simulation, we train two decision tree classifiers: one using the full dataset, and a second, referred to as the "weaker" model, using only 50% of the same dataset. The decision tree trained on the full dataset is assumed to produce the "expected" test outcomes for any given input. The aim of this simulation is to assess how effectively the generated test inputs challenge the weaker model, thereby revealing its reduced accuracy due to limited training data. The fully trained decision tree is provided to our framework as the input data-driven model, from which test inputs are subsequently generated. These inputs are executed on both the full and weaker decision trees, producing two sets of test outputs. We compare the expected outputs from the full model with the actual outputs from the weaker model using the F_1 score. The F_1 score is a widely adopted metric for evaluating the performance of classification models, particularly in scenarios involving imbalanced class distributions [6]. While alternative metrics exist, we consider the F_1 score the most suitable for evaluating the effectiveness of test inputs, as it captures the ability of a model to correctly identify true positives while minimising false positives and false negatives. By comparing F_1 scores across different test input generation strategies, we assess how effectively each strategy exposes both correct and incorrect classifications made by the weaker model.

In the second simulation, we follow a similar procedure using neural networks. One network is trained on the full dataset, while the weaker network is trained on 50% of the same data. As before, the full model is assumed to produce the expected test outcomes. The goal is to evaluate how well the generated test inputs expose the limitations of the weaker neural network. The fully trained neural network is provided to our framework as the input data-driven model. The framework applies surrogate modelling, as described in Section 4.3, to learn a decision tree that approximates the behaviour of the neural network. Test inputs are generated from this surrogate model and executed on both the full and weaker neural networks, yielding two sets of test outputs. We compare these outputs using the MAE, a standard metric for evaluating the performance of regression models. Although other metrics are available, MAE is particularly suitable here as it quantifies the average magnitude of error between expected and actual outputs. By comparing MAE values across different test input generation strategies, we determine how effectively each strategy reveals prediction errors in the weaker model.

We apply the first simulation to all datasets used for classification tasks, as listed in Table 4, and the second simulation to those used for regression tasks, shown in Table 5. For each simulation run, the framework is executed four times, each with a distinct test requirement (TR):

- **TR1:** DTC (Definition 4.1) with the proportional test allocation strategy (Definition 4.3), referred to as DTC Proportional.
- **TR2:** DTC with the inverse test allocation strategy (Definition 4.4), referred to as DTC Inverse.
- **TR3:** BVA (Definition 4.2) with the proportional test allocation strategy, referred to as BVA Proportional.
- **TR4:** BVA with the inverse test allocation strategy, referred to as BVA Inverse.

In all cases, the framework is configured to generate a total of 5000 test inputs. The global range for each feature is determined using the observed minimum and maximum values within the respective dataset, rather than relying on manually specified feature ranges. For BVA-based executions, the neighbourhood size around boundary values is set to $\epsilon = 1.0$.

In addition to our framework, we generate test inputs for each dataset using four baseline methods. This results in four distinct sets of test inputs per dataset. For classification datasets, these inputs are executed on both the fully trained and weaker decision trees. For regression datasets, they are executed on the corresponding full and weaker neural networks. Each execution yields a set of expected and actual test outputs. Depending on the task type, we compute either the F_1 score or MAE for each pair of outputs. We compare these metrics across all test input generation strategies to assess the relative effectiveness of the generated test inputs.

Baselines. We present four baseline methods for generating test inputs, which serve as reference points for evaluating those produced by our framework.

1. *Random Sampling* (RS): Samples 5000 random values from the global range of each input feature, defined by the minimum and maximum values observed in the corresponding dataset. These values are then combined across features to form 5000 test inputs.
2. *Boundary Sampling* (BS): Samples 5000 values from the neighbourhood around the upper and lower bounds of each global feature range, using a neighbourhood size of $\epsilon = 1.0$. These values are then combined across features to form 5000 test inputs.

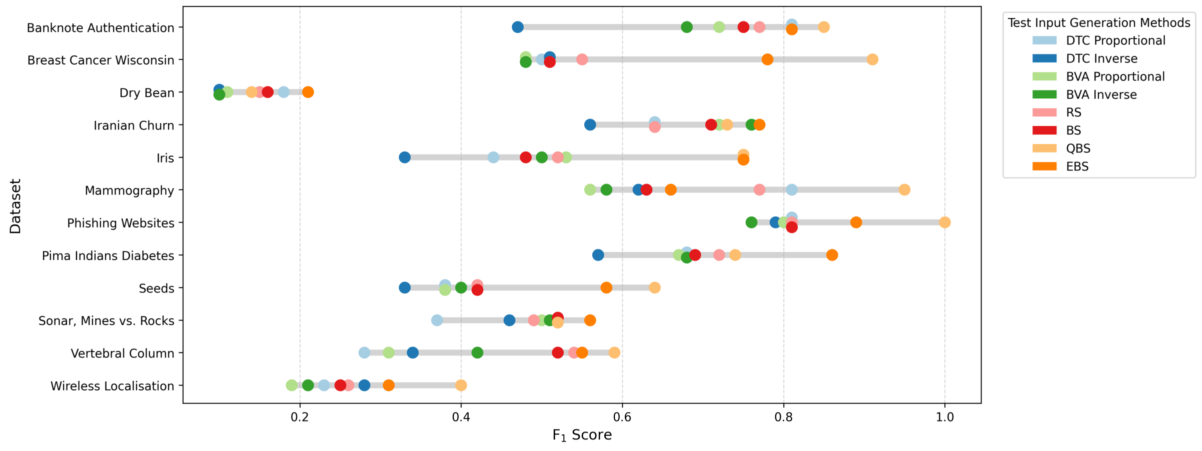


Figure 8: Effectiveness comparison of test inputs generated by our framework and baseline methods in the first simulation. The x-axis shows the F_1 score, and the y-axis lists the datasets used for classification tasks (see Table 4). Each dot represents the F_1 score of test inputs generated either by our framework using one of four test requirements (DTC Proportional, DTC Inverse, BVA Proportional, BVA Inverse) or by one of four baseline methods (RS, BS, QBS, EBS).

3. *Quantile-Based Sampling* (QBS): Each feature range is divided into four intervals, also known as quantiles, each containing an equal portion of values from the dataset. From each quantile, 1250 synthetic values are generated and combined across features to form 5000 test inputs.
4. *Example-Based Sampling* (EBS): Samples 5000 examples directly from the dataset to use as test inputs. Sampling is performed with replacement if the dataset contains fewer than 5000 instances, and without replacement otherwise.

The first three baseline methods employ random sampling with varying levels of complexity, depending on how the sampling intervals are defined. RS is the simplest, relying on global feature ranges, whereas QBS is the most complex, using quantile-based partitions for each feature range. These methods are conceptually aligned with our framework, which also applies random sampling within equivalence class intervals and near their boundaries. The final baseline is a widely adopted strategy for generating test inputs to evaluate machine learning models, commonly used in techniques such as cross-validation and bootstrapping [17].

7. Evaluation

In this section, we analyse the simulation results comparing our framework against four baseline methods, with reference to the formulated hypotheses. We then evaluate the overall performance of our framework in generating effective test inputs and examine its limitations. Finally, we discuss potential threats to the validity of our research.

7.1. Hypothesis Testing

7.1.1. Effectiveness of Test Inputs Without Surrogate Modeling

Figure 8 presents the results of the first simulation using a connected dot plot. Each horizontal line corresponds to a classification dataset (see Table 4), and each dot represents the F_1 score obtained from a distinct set of test inputs executed on decision trees trained on that dataset. For each dataset, eight F_1 scores are shown. Four come from our framework under different test requirements, and four from the baseline methods (Section 6.3). Identical scores within a dataset are stacked vertically with a slight offset.

Lower F_1 scores (dots positioned further left) indicate higher effectiveness in revealing model misclassifications, as defined in Definition 6.1. Across all datasets, at least one configuration from our

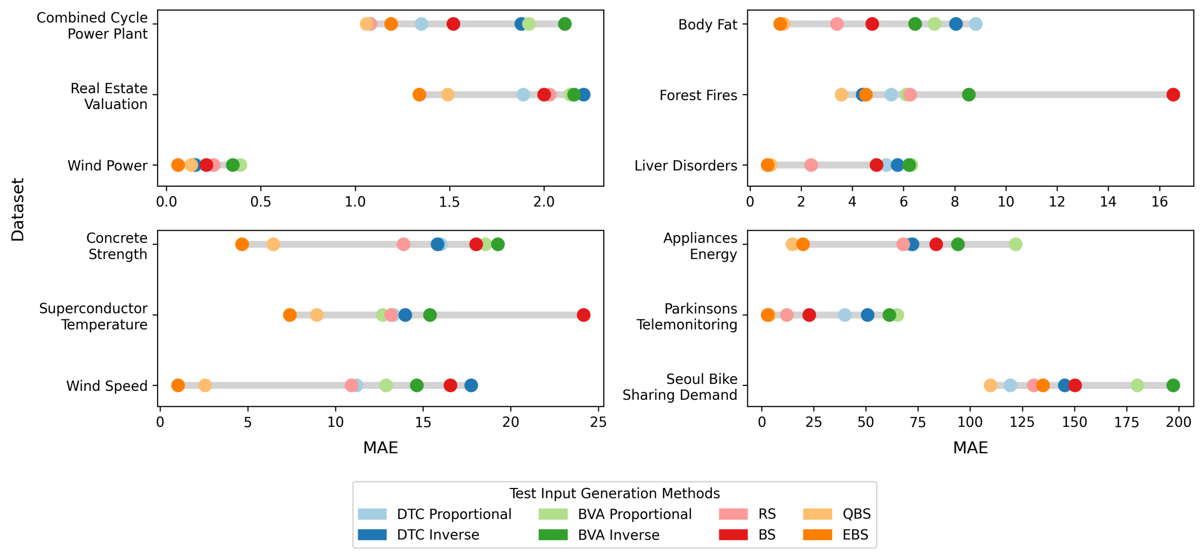


Figure 9: Effectiveness comparison of test inputs generated by our framework and baseline methods in the second simulation. The figure consists of four plots, each displaying results for datasets with similar MAE scales. In all plots, the x-axis shows the MAE, and the y-axis lists the datasets used for regression tasks (see Table 5). Each dot represents the MAE obtained from test inputs generated either by our framework using one of four test requirements (DTC Proportional, DTC Inverse, BVA Proportional, BVA Inverse) or by one of four baseline methods (RS, BS, QBS, EBS).

framework outperforms the baselines. Notably, no configuration performs worse than all baseline methods on any dataset.

However, performance varies across test requirements and datasets. This is due to how test inputs are allocated across different methods. Baseline methods sample uniformly across the global feature range or its partitions, giving equal chance to discover effective inputs. In contrast, our framework allocates test inputs based on the distribution of training examples across the leaves of decision trees. When effective test inputs lie in dominant classes, DTC Inverse may overlook them, whereas DTC Proportional is more likely to detect them. This pattern is evident in datasets such as "Wireless Localisation", where DTC Inverse configuration performs less effectively, but DTC Proportional consistently outperforms all baselines. Conversely, when effective inputs lie in less dominant classes, DTC Proportional may underperform, as seen in "Mammography", where DTC Inverse proves more effective.

A similar pattern is observed in BVA configurations. When effective inputs are near boundaries of dominant classes, BVA Inverse may miss them, while BVA Proportional is more likely to detect them. The reverse also occurs, where BVA Inverse outperforms BVA Proportional.

In "Solar, Mines vs Rocks", BVA configurations perform worse than RS, likely because effective test inputs lie within feature intervals rather than near boundaries. In such cases, DTC configurations yield better results. The opposite is also observed, where BVA configurations outperform other methods. This suggests that effective inputs lie near equivalence class boundaries rather than within intervals.

Overall, the simulation results partially support Hypothesis 1. Without surrogate modelling, our framework can generate more effective test inputs than baseline methods, but only when configured to target fault-prone regions of the input space. Under suitable test requirements, our approach can produce challenging inputs that expose system weaknesses beyond what simple random sampling achieves.

7.1.2. Effectiveness of Test Inputs With Surrogate Modeling

Figure 9 presents the results of the second simulation using four connected dot plots. Each plot groups datasets with similar MAE scales. As in the first simulation, each horizontal line represents a regression dataset (see Table 5), and each dot shows the MAE score from test inputs generated either by our

Table 6
Volumes of the MBB computed from all sets of test inputs across both simulations. Test inputs are generated either by our framework using one of four test requirements (DTC Proportional, DTC Inverse, BVA Proportional, BVA Inverse) or by one of four baseline methods (RS, BS, QBS, EBS).

Dataset	DTC Proportional	DTC Inverse	BVA Proportional	BVA Inverse	RS	BS	QBS	EBS
Banknote Authentication	94 409	94 382	59 908	58 386	94 380	149 313	81 539	94 609
Breast Cancer Wisconsin	5.60×10^{8}	5.61×10^{8}	2.42×10^{14}	1.40×10^{14}	5.56×10^{8}	2.63×10^{29}	2.56×10^{8}	5.63×10^{8}
Dry Bean	1.76×10^{13}	1.69×10^{13}	2.90×10^{24}	2.18×10^{24}	1.75×10^{13}	3.50×10^{25}	7.86×10^{12}	1.11×10^{13}
Iranian Churn	1.21×10^{21}	1.20×10^{21}	1.21×10^{21}	9.60×10^{20}	1.21×10^{21}	7.05×10^{22}	0	1.20×10^{21}
Iris	132	131	150	148	122	855	114	122
Mammography	3.91×10^{6}	3.81×10^{6}	8.05×10^{5}	8.14×10^{5}	3.90×10^{6}	1.32×10^{7}	3.08×10^{6}	1.99×10^{6}
Phishing Websites	5.37×10^{8}	5.37×10^{8}	2.68×10^{8}	2.68×10^{8}	5.37×10^{8}	8.65×10^{17}	0	5.37×10^{8}
Pima Indians Diabetes	4.35×10^{14}	4.32×10^{14}	2.31×10^{14}	2.28×10^{14}	4.35×10^{14}	1.00×10^{15}	3.45×10^{14}	4.35×10^{14}
Seeds	220	220	10 784	9832	219	90 876	172	220
Sonar, Mines vs. Rocks	1.16×10^{-22}	1.16×10^{-22}	2.67×10^{-11}	2.84×10^{-11}	1.08×10^{-22}	9.73×10^{24}	1.44×10^{-23}	1.10×10^{-22}
Vertebral Column	2.87×10^{12}	2.85×10^{12}	2.40×10^{11}	2.36×10^{11}	2.79×10^{12}	3.14×10^{12}	2.43×10^{12}	2.80×10^{12}
Wireless Localisation	2.70×10^{11}	2.70×10^{11}	4.57×10^{10}	4.38×10^{10}	2.70×10^{11}	3.77×10^{11}	2.11×10^{11}	2.62×10^{11}
Appliances Energy	1.18×10^{40}	1.04×10^{40}	3.59×10^{38}	3.29×10^{38}	2.09×10^{38}	2.42×10^{40}	8.20×10^{37}	1.31×10^{38}
Body Fat	1.52×10^{20}	1.82×10^{20}	4.76×10^{18}	4.31×10^{18}	6.69×10^{18}	3.72×10^{20}	3.52×10^{18}	6.73×10^{18}
Combined Cycle Power Plant	2.42×10^{8}	2.45×10^{8}	2.88×10^{7}	2.84×10^{7}	5.97×10^{6}	7.05×10^{8}	5.42×10^{6}	5.71×10^{6}
Concrete Strength	8.30×10^{19}	8.86×10^{19}	2.75×10^{15}	2.67×10^{15}	6.32×10^{18}	1.39×10^{20}	4.18×10^{18}	6.34×10^{18}
Forest Fires	1.75×10^{16}	1.72×10^{16}	2.66×10^{14}	2.61×10^{14}	9.15×10^{15}	3.21×10^{25}	0	9.17×10^{15}
Liver Disorders	1.66×10^{10}	1.63×10^{10}	9.54×10^{8}	9.54×10^{8}	1.48×10^{10}	5.47×10^{10}	1.21×10^{10}	1.48×10^{10}
Parkinsons Telemonitoring	1.04×10^{-2}	1.02×10^{-2}	5.33×10^{-5}	5.26×10^{-5}	3.41×10^{-6}	6.00×10^{11}	2.11×10^{-6}	3.22×10^{-6}
Real Estate Valuation	19 792	19 945	14	14	19 798	4.51×10^{7}	16 465	19 849
Superconductor Temperature	3.01×10^{160}	2.90×10^{160}	8.98×10^{158}	8.31×10^{158}	1.00×10^{160}	3.37×10^{166}	6.36×10^{158}	4.66×10^{158}
Seoul Bike Sharing Demand	3.57×10^{14}	3.50×10^{14}	4.27×10^{10}	4.19×10^{10}	1.18×10^{14}	2.64×10^{17}	0	1.14×10^{14}
Wind Power	1.18×10^{15}	1.18×10^{15}	3.82×10^{14}	3.83×10^{14}	8.50×10^{14}	2.24×10^{15}	7.23×10^{14}	3.67×10^{14}
Wind Speed	4.15×10^{7}	4.11×10^{7}	1.76×10^{6}	1.00×10^{6}	3.73×10^{7}	2.36×10^{8}	2.55×10^{7}	2.48×10^{7}

framework or by the baseline methods.

Unlike the F_1 score, lower MAE values (dots positioned further to the left) indicate lower effectiveness in revealing prediction errors, as defined in Definition 6.2. In ten out of twelve datasets, at least one configuration from our framework outperforms the baselines. However, the most effective configuration varies across datasets due to differences in test input allocation strategy employed by each method, as discussed earlier. As also seen in the first simulation, the relative performance of DTC and BVA configurations depends on whether effective test inputs lie near equivalence class boundaries or within feature intervals.

A pattern not observed in the first simulation is that, for "Superconductor Temperature" and "Forest Fires", none of the configurations from our framework outperform BS. In both datasets, BS achieves significantly better results. This is likely because the most effective inputs are located at the boundaries of the global feature range. Our framework, particularly under BVA configurations, does not typically generate inputs from these regions, as such boundaries are rarely used in the split conditions of decision trees. Instead, it targets equivalence class boundaries, which usually lie within the feature range.

The second simulation results are partially consistent with Hypothesis 2. Our framework can outperform the baseline methods even when surrogate modelling is applied, given that the test requirements are appropriately configured. However, its performance declines when the most effective inputs are concentrated solely at the global boundaries.

7.1.3. Input Space Coverage

Table 6 presents the volumes of the MBB (see Definition 6.3), calculated from all sets of test inputs across both simulations. These volumes indicate the extent of input space coverage achieved by each set. The test inputs are generated either by our framework or by one of four baseline methods. Based on these results, we select eight representative datasets and visualise their volumes in Figure 10. Each chart in the figure displays the corresponding MBB volume on a logarithmic scale.

Across all datasets, BS consistently achieves the highest input space coverage. This is because BS samples test inputs around both the lower and upper bounds of each global feature range. Since all methods operate over a uniform global range, BS inherently covers the full input space by design.

The DTC and BVA configurations yield identical input space coverage within their respective pairs. This behaviour is consistent with the design, as each pair samples from the same equivalence class

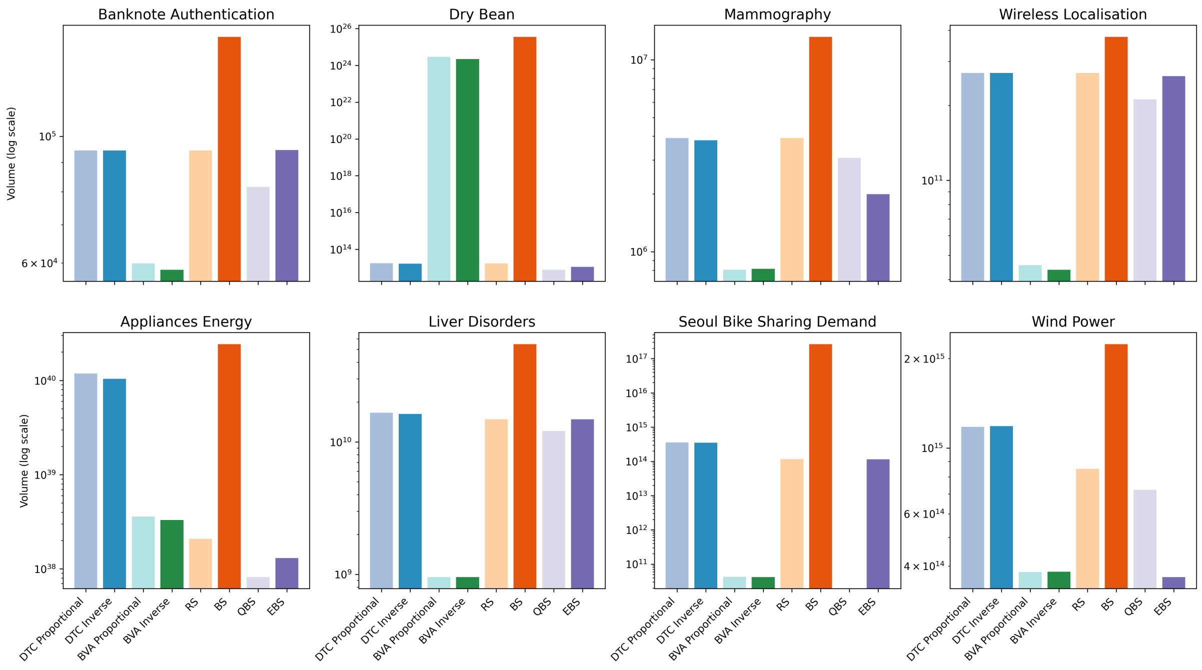


Figure 10: Comparison of input space coverage by different sets of test inputs across both simulations. The figure comprises eight bar charts, showing results from representative datasets selected to highlight key patterns and differences. In every plot, bars indicate the volume of the MBB (as defined in Definition 6.3), displayed on a logarithmic scale.

structure, either within the intervals or around the boundaries, but applies different allocation strategies to distribute test inputs.

DTC configurations generally achieve comparable or significantly greater coverage than RS, QBS, and EBS. RS and EBS rely on random sampling, which may fail to include test inputs spanning the entire space. While QBS divides the global feature range into quantiles and samples evenly across them, our framework leverages the internal structure of decision trees to define equivalence classes. This enables more granular and systematic coverage of the input space.

BVA configurations typically result in lower coverage than DTC. This is because BVA targets inputs only around equivalence class boundaries defined by the split conditions of the decision tree. As a result, test inputs may be generated around either the lower or upper bounds of intervals, but not both, leading to reduced overall coverage. An exception is observed in "Dry Bean", where BVA configurations outperform all methods except BS and achieve coverage comparable to BS. This is likely due to the decision tree defining boundaries near both ends of the global feature range, allowing BVA to sample more broadly.

These results partially support Hypothesis 3. In most cases, DTC configurations within our framework generate test inputs that cover a larger portion of the input space than RS, QBS, and EBS. BVA configurations can achieve high coverage only when decision tree splits are positioned near both ends of the global feature range.

7.1.4. Dispersion of Test Inputs

Figure 11 presents a connected dot plot comparing the CV across test input sets from both simulations. Each horizontal line represents a distinct dataset, with dots indicating the CV calculated from inputs generated either by our framework or by one of the baseline methods.

The CV quantifies the relative variability of a distribution with respect to its mean, indicating how test inputs are spread within the covered input space. As defined in Definition 6.4, a higher CV (dots positioned further to the right) reflects greater variability, suggesting that test inputs are more widely scattered across different regions of the input space.

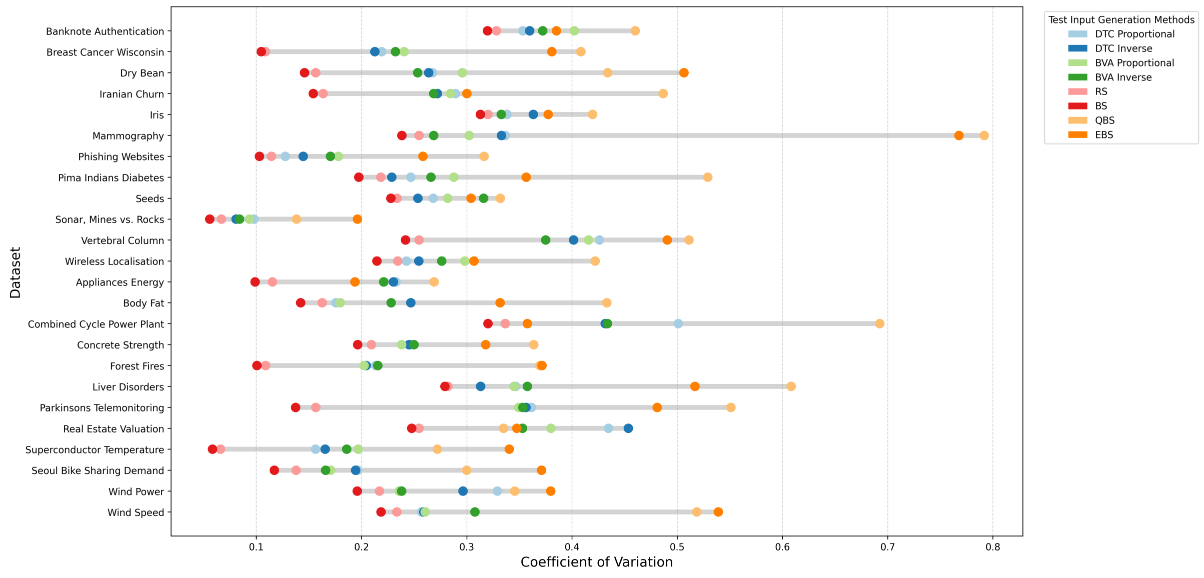


Figure 11: Comparison of the dispersion of test inputs within the covered input space across both simulations. The x-axis shows the CV, and the y-axis lists the datasets. Each dot represents the CV for a given dataset, calculated from test inputs generated either by our framework, using one of four test requirements (DTC Proportional, DTC Inverse, BVA Proportional, BVA Inverse) or by one of four baseline methods (RS, BS, QBS, EBS).

From the figure, we observe that BS consistently underperforms compared to other methods. BS generates inputs only near the boundaries of each global feature range. While this may produce a balanced distribution around the mean, it leaves large portions of the input space untested, resulting in low variability and a lower CV.

RS yields the second-lowest CV values among all methods. Although it samples uniformly across each global feature range, the resulting test inputs tend to cluster around the mean, leaving boundary regions sparsely populated [30]. This effect, explained by the law of large numbers, becomes more evident in high-dimensional spaces due to the curse of dimensionality. As a result, RS exhibits lower variability, indicating that its test inputs are less dispersed across the input space. The effect extends across datasets, as those with more features (i.e., higher-dimensional input spaces) tend to yield lower CV values under RS compared to datasets with fewer features.

Our framework consistently produces higher CVs than both RS and BS. This is because it partitions the input space into equivalence classes and samples test inputs from each class or around their boundaries. This strategy ensures broader coverage and greater dispersion, leading to more scattered inputs across the input space.

However, our framework does not outperform QBS in most datasets. This is due to differences in the test allocation strategy. Our framework allocates more inputs to either dominant or less dominant equivalence classes, depending on the chosen strategy. This causes clustering around specific regions, shifting the mean and reducing variability. Although inputs are still sampled from other classes, they are included less frequently. This contributes to increased variability, but not to the extent achieved by QBS. In contrast, QBS selects inputs from evenly spaced quantiles across the entire feature range, resulting in consistent dispersion and a higher CV. If training examples are distributed evenly across the leaf nodes, our framework generates a similar number of inputs per equivalence class. In such cases, it can yield a higher CV than QBS.

EBS generally yields high CV values, though its performance varies across datasets. For example, in "Combined Cycle Power Plant", EBS produces a lower CV than our framework. This variability stems from EBS sampling directly from the dataset, meaning the dispersion of test inputs reflects the underlying data distribution. If the dataset is widely scattered across the input space, the resulting CV will be high. If the data is concentrated in a specific region, the CV will be lower.

These findings support Hypothesis 4. Test inputs generated by our framework are scattered across the covered input space in a way that aligns with the proposed test allocation strategies. This is shown by their CV values, which are consistently higher than those of BS and RS, but lower than those achieved by QBS.

7.2. Discussion

The simulation results show that, across all evaluation criteria, the most effective test configuration within our framework varies depending on the dataset. This suggests that, while the framework is applicable to a range of data-driven models, it also exhibits flexibility. Specifically, by adjusting the test requirements, the framework can be tailored to generate test inputs that target fault-prone regions within the input domain of each SUT. These targeted inputs help uncover potential weaknesses in the system. However, if such regions lie near the boundaries of the global feature range, the framework may fail to produce inputs that effectively expose them, allowing certain faults to go undetected.

An aspect not reflected in the simulation results is the framework's dependence on the quality of the data-driven models used to generate test inputs. For the framework to perform effectively, the data-driven models must either be decision trees with split conditions that accurately predict output labels at the leaves, or models that can be reliably approximated by tree surrogates with equally precise split conditions. These split conditions are essential for defining equivalence classes used in sampling test inputs. If the splits are poorly defined, the resulting classes may fail to represent meaningful regions of the input space that are processed similarly by the SUT. In such cases, fault-prone areas may not be properly isolated, and test inputs drawn from these classes may fail to reveal system weaknesses.

Another limitation is the framework's inability to distinguish between time-series and non-time series data. Although time-series patterns are common in CPSs, the framework does not account for explicit model specifications related to temporal behaviour. It treats all input data as independent, ignoring any time-dependent structure. To ensure reliable test input generation, the underlying data-driven model must first transform any existing time-series dynamics into independent, non-sequential data before being passed to the framework.

Despite these considerations, the framework offers a reliable and structured approach to system evaluation, allowing thorough exploration of system boundaries and limitations. This is particularly useful in CPSs, where edge cases and uncommon scenarios can have a significant impact on their performance [5].

One practical use of the framework is in regression testing. As the system evolves through software updates, hardware modifications, or environmental shifts, it is crucial to ensure that existing functionality is preserved. The framework allows testers to generate inputs from a nominal system and execute them on both the original and updated systems. By comparing the test outputs using a common metric, testers can determine whether stability or performance of the SUT has been affected by the changes.

7.3. Threats to Validity

One potential threat to external validity is that the datasets used in our simulations may not reflect the input-output behaviour of real-world CPSs. These datasets are typically cleaner and less complex than those encountered in practice. As a result, it is possible that different outcomes might have observed had we used larger, more complex datasets collected from actual physical systems.

A threat to construct validity arises from the possibility that the evaluation metrics employed may not fully capture the effects we intended to investigate [17]. Although F1 score and MAE are standard measures for assessing machine learning models, they may not adequately reflect the quality of test inputs in the context of CPSs. Nonetheless, these metrics provide a clear and interpretable means of evaluating how effectively the inputs challenge the SUT, particularly in exploring edge cases and uncommon scenarios.

Other metrics could potentially be used to evaluate input space coverage and the dispersion of test inputs, besides bounding box volume and the CV. However, these two align well with the purpose of our analysis. Bounding box volume indicates how much of the input space is covered, while CV shows how spread out the inputs are within that space. Their definitions and calculation methods make them suitable choices for assessing these aspects.

Finally, another threat to construct validity may stem from potential faults in the implementation of our framework. Any such issues could affect the accuracy of the results and the reliability of the conclusions drawn.

8. Concluding Remarks

Cyber-Physical Systems (CPSs) are inherently large-scale and complex due to their integration of computing, communication, and control technologies. Rigorous testing is essential prior to deployment to ensure system reliability and safety. However, testing CPSs is challenging due to the difficulty of translating abstract requirements into concrete test inputs and accurately representing complex system behaviour.

To address these challenges, we introduce a test input generation framework based on data-driven models of CPSs. These models, constructed from system-generated data, reflect the behaviour of the physical system. By utilising data-driven models, the framework removes the burden of manually analysing or interpreting abstract requirements, as such requirements are implicitly encoded within the data. Furthermore, data-driven modelling eliminates the need for specific domain expertise, making the testing process more accessible. Our framework employs decision trees as white-box models to guide test input generation. Specifically, it partitions the input domain of the SUT into equivalence classes, derived from the tree's internal structure. Test inputs are then statistically sampled from each class based on test requirements specified by the users. Additionally, we propose a method for training a decision tree surrogate from a data-driven model, allowing the same testing strategy to be applied across different model types.

Simulation results show that, by adjusting the test requirements, our framework can be flexibly adapted to each SUT, allowing the generation of challenging inputs that effectively stress the system. These inputs span a broad range of values and are strategically dispersed throughout the input space, ensuring targeted and comprehensive exploration.

The framework generates test inputs offline, meaning all inputs are created in advance based on the test requirements before execution on the SUT. Offline generation offers several advantages: it allows for faster execution and enables pre-analysis of test coverage [31]. As part of future work, we explore model-based online testing, which combines input generation and execution in real time. In this approach, the test generator interactively stimulates and observes the SUT. Only a single input is generated from the model at a time, which is immediately executed on the SUT, and the resulting output is checked against the system specification. This process repeats until the test concludes or an error is detected. The method supports dynamic adaptation of test requirements and facilitates the automatic identification of fault-prone regions within the input space that require further exploration. However, online testing is more complex, particularly for CPSs, as it demands interpretation of system specifications and precise output evaluation. This requires embedded domain knowledge, which is difficult to integrate due to the complex and technically diverse nature of CPSs, involving real-time constraints and tightly coupled physical and digital components.

In addition to online testing, we plan to investigate alternative white-box surrogate models for test input generation. One promising candidate is the random forest, a machine learning model that aggregates the outputs of multiple decision trees to produce a single prediction [32]. Random forests offer multiple options for partitioning the input space into equivalence classes. This flexibility allows us to select the most meaningful partitioning, where each class accurately represents a region of the input space processed similarly by the SUT. Moreover, by combining equivalence class partitions across different trees within the forest, the framework can construct an optimal structure for generating highly effective test inputs. This direction opens new possibilities for enhancing the precision and adaptability of our framework.

Declaration on Generative AI

During the preparation of this work, the authors used Microsoft Copilot in order to: Grammar and spelling check. After using the tool, the authors reviewed and edited the content as needed and takes full responsibility for the publication's content.

References

[1] M. Schmidt, S. Plambeck, M. Knitt, H. Rose, G. Fey, J. C. Wieck, S. Balduin, Flowcean — model learning for cyber-physical systems, 4th Italian Workshop on Artificial Intelligence and Applications for Business and Industries - AIABI (2024).

[2] K. Kim, P. R. Kumar, Cyber-physical systems: A perspective at the centennial, Proc. IEEE 100 (2012) 1287–1308. doi:10.1109/JPROC.2012.2189792.

[3] R. Rajkumar, I. Lee, L. Sha, J. A. Stankovic, Cyber-physical systems: the next computing revolution, in: S. S. Sapatnekar (Ed.), Proceedings of the 47th Design Automation Conference, DAC 2010, Anaheim, California, USA, July 13-18, 2010, ACM, 2010, pp. 731–736. doi:10.1145/1837274.1837461.

[4] A. Aerts, M. Reniers, M. R. Mousavi, Model-based testing of cyber-physical systems, in: Cyber-physical systems, Elsevier, 2017, pp. 287–304.

[5] Z. Sadri-Moshkenani, J. M. Bradley, G. Rothermel, Survey on test case generation, selection and prioritization for cyber-physical systems, Softw. Test. Verification Reliab. 32 (2022). doi:10.1002/STVR.1794.

[6] A. Burkov, The hundred-page machine learning book, volume 1, Andriy Burkov Quebec City, QC, Canada, 2019.

[7] S. Plambeck, G. Fey, Data-driven test generation for black-box systems from learned decision tree models, in: M. Jenihhin, H. Kubátová, N. Metens, J. Raik, F. Ahmed, J. Belohoubek (Eds.), 26th International Symposium on Design and Diagnostics of Electronic Circuits and Systems, DDECS 2023, Tallinn, Estonia, May 3-5, 2023, IEEE, 2023, pp. 27–32. doi:10.1109/DDECS57882.2023.10139633.

[8] Y.-Y. Song, Y. Lu, Decision tree methods: applications for classification and prediction, Shanghai archives of psychiatry (2015).

[9] P. M. Domingos, G. Hulten, Mining high-speed data streams, in: R. Ramakrishnan, S. J. Stolfo, R. J. Bayardo, I. Parsa (Eds.), Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining, Boston, MA, USA, August 20-23, 2000, ACM, 2000, pp. 71–80. doi:10.1145/347090.347107.

[10] J. V. Deshmukh, M. Horvat, X. Jin, R. Majumdar, V. S. Prabhu, Testing cyber-physical systems through bayesian optimization, ACM Trans. Embed. Comput. Syst. 16 (2017) 170:1–170:18. doi:10.1145/3126521.

[11] C. Menghi, S. Nejati, L. C. Briand, Y. I. Parache, Approximation-refinement testing of compute-intensive cyber-physical models: an approach based on system identification, in: G. Rothermel, D. Bae (Eds.), ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020, ACM, 2020, pp. 372–384. doi:10.1145/3377811.3380370.

[12] A. M. Kosek, O. Gehrke, Ensemble regression model-based anomaly detection for cyber-physical intrusion detection in smart grids, in: 2016 IEEE Electrical Power and Energy Conference (EPEC), 2016, pp. 1–7. doi:10.1109/EPEC.2016.7771704.

[13] J. M. Carter, L. Lin, J. H. Poore, Automated functional testing of simulink control models, in: Proceedings of the 1st workshop on model-based testing in practice, Berlin, Germany, Citeseer, 2008, pp. 41–50.

[14] H. L. da Silva Araujo, G. Carvalho, M. R. Mousavi, A. Sampaio, Multi-objective search for effective testing of cyber-physical systems, in: P. C. Ölveczky, G. Salaün (Eds.), Software Engineering and Formal Methods - 17th International Conference, SEFM 2019, Oslo, Norway, September 18-20, 2019, Proceedings, volume 11724 of *Lecture Notes in Computer Science*, Springer, 2019, pp. 183–202. doi:10.1007/978-3-030-30446-1_10.

[15] L. Zhang, J. He, W. Yu, Test case generation from formal models of cyber physical system, International Journal of Hybrid Information Technology 6 (2013) 15–24.

[16] B. Cukic, B. J. Taylor, H. Singh, Automated generation of test trajectories for embedded flight control systems, Int. J. Softw. Eng. Knowl. Eng. 12 (2002) 175–200. doi:10.1142/S0218194002000895.

[17] V. H. S. Durelli, R. Monteiro, R. S. Durelli, A. T. Endo, F. C. Ferrari, S. R. S. Souza, Property-based testing for machine learning models, in: E. L. G. Alves, M. Ribeiro (Eds.), Proceedings of the 9th Brazilian Symposium on Systematic and Automated Software Testing, SAST 2024, Curitiba, Brazil, September 30 - October 4, 2024, Sociedade Brasileira de Computação, 2024, pp. 39–48. doi:10.5753/SAST.2024.3791.

[18] A. Sharma, C. Demir, A. N. Ngomo, H. Wehrheim, MLCHECK- property-driven testing of machine learning classifiers, in: M. A. Wani, I. K. Sethi, W. Shi, G. Qu, D. S. Raicu, R. Jin (Eds.), 20th IEEE International Conference on Machine Learning and Applications, ICMLA 2021, Pasadena, CA, USA, December 13-16, 2021, IEEE, 2021, pp. 738–745. doi:10.1109/ICMLA52953.2021.00123.

[19] A. Aggarwal, P. Lohia, S. Nagar, K. Dey, D. Saha, Black box fairness testing of machine learning models, in: M. Dumas, D. Pfahl, S. Apel, A. Russo (Eds.), Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019, ACM, 2019, pp. 625–635. doi:10.1145/3338906.3338937.

[20] O. Biran, C. Cotton, Explanation and justification in machine learning: A survey, in: IJCAI-17 workshop on explainable AI (XAI), volume 8, 2017, pp. 8–13.

[21] R. Black, Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional, John Wiley & Sons, Inc., USA, 2007.

[22] A. Sweigart, The Recursive Book of Recursion: Ace the Coding Interview with Python and JavaScript, No Starch Press, 2022.

[23] IEEE standard for floating-point arithmetic, IEEE Std 754-2008 (2008) 1–70. doi:10.1109/IEEESTD.2008.4610935.

[24] W. Nash, T. Sellers, S. Talbot, A. Cawthorn, W. Ford, Abalone, UCI Machine Learning Repository, 1994. doi:10.24432/C55C7W.

[25] O. Boz, Converting A trained neural network to a decision tree dectext - decision tree extractor, in: M. A. Wani, H. R. Arabnia, K. J. Cios, K. Hafeez, G. Kendall (Eds.), Proceedings of the 2002 International Conference on Machine Learning and Applications - ICMLA 2002, June 24-27, 2002, Las Vegas, Nevada, USA, CSREA Press, 2002, pp. 110–116.

[26] B. W. Silverman, Density Estimation for Statistics and Data Analysis, Springer, 1986. doi:10.1007/978-1-4899-3324-9.

[27] E. Evans, Domain-Driven Design: Tacking Complexity In the Heart of Software, Addison-Wesley Longman Publishing Co., Inc., USA, 2003.

[28] G. T. Toussaint, Solving geometric problems with the rotating calipers, in: Proc. IEEE Melecon, volume 83, 1983, p. A10.

[29] B. S. Everitt, A. Skrondal, The Cambridge dictionary of statistics, volume 4, Cambridge university press Cambridge, UK, 2010.

[30] C. Giraud, Introduction to high-dimensional statistics, Chapman and Hall/CRC, 2021.

[31] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, A. Skou, Testing real-time systems using UPPAAL, in: R. M. Hierons, J. P. Bowen, M. Harman (Eds.), Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers, volume 4949 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 77–117. doi:10.1007/978-3-540-78917-8_3.

[32] L. Breiman, Random forests, Mach. Learn. 45 (2001) 5–32. doi:10.1023/A:1010933404324.