Mining Assumptions for Software Components using Machine Learning

Anonymous Author(s)

ABSTRACT

Software verification approaches aim to check a software component under analysis for all possible environments. In reality, however, components are expected to operate within a larger system and are required to satisfy their requirements only when their inputs are constrained by *environment assumptions*. In this paper, we propose EPIcuRus, an approach to automatically synthesize environment assumptions for a component under analysis (i.e., conditions on the component inputs under which the component is guaranteed to satisfy its requirements). EPIcuRus combines search-based testing, machine learning and model checking. The core of EPIcuRus is a decision tree algorithm that infers environment assumptions from a set of test results including test cases and their verdicts. The test cases are generated using search-based testing, and the assumptions inferred by decision trees are validated through model checking. In order to improve the efficiency and effectiveness of the assumption generation process, we propose a novel test case generation technique, namely Important Features Boundary Test (IFBT), that guides the test generation based on the feedback produced by machine learning. We evaluated EPIcuRus by assessing its effectiveness in computing assumptions on a set of study subjects that include 18 requirements of four industrial models. We show that, for each of the 18 requirements, EPIcuRus was able to compute an assumption to ensure the satisfaction of that requirement, and further, ≈ 78% of these assumptions were computed in one hour.

1 INTRODUCTION

The ultimate goal of software verification techniques is to provably demonstrate the correctness of the system under analysis for a given system requirement. However, this problem is generally undecidable for cyber-physical or hybrid systems [38], and too difficult to achieve for many industrial systems that are far too complex to be verified exhaustively and in their entirety [42, 54]. As a result, in practice, exhaustive verification techniques (e.g., [6, 33, 62]) can only be applied to some selected individual components or algorithms that are amenable to exhaustive verification and are used within a larger system.

In general, we cannot conclude if an entire system satisfies or refutes some requirements by verifying the constituent components of that system. In other words, the verification results of individual components cannot be generally lifted to the system level. Unless we rely on some compositional verification framework, component verification alone does not contribute to evaluating a system as a whole. However, exhaustively verifying individual components can still be beneficial, especially for components performing some critical algorithmic computation or components that are reused in several systems. In checking components individually, however, it is essential to identify the environment information in which each component is expected to operate correctly [34]. Attempting to verify components for a more general environment than their expected operational environment may lead to overly pessimistic results or to detecting spurious failures.

The problem of identifying the environmental context for individual software components has been extensively studied by classical compositional and assume-guarantee reasoning approaches (e.g., [1, 27, 30, 39]). These approaches capture the context for an individual software component by describing the expectations that a component makes about its environment using *environment assumptions* (i.e., conditions or constraints on component inputs). Traditional compositional verification frameworks expect environment assumptions to be developed manually. This is, however, difficult and time-consuming. For most practical cases, engineers simply may not have sufficient information about the details of the components under analysis to develop a sufficiently detailed, useful and accurate environment assumption. There have been approaches to automate environment assumption computation in the context of assume-guarantee reasoning using an exact learning algorithm for regular languages and finite state automata [24, 34, 36]. These approaches, however, assume that the components under analysis and their environment can be specified as abstract finite-state machines. State machines are not, however, expressive enough to capture quantitative and numerical components as well as continuous behavior of systems interacting with physical environments. Besides, software components may not be readily specified in (abstract) state machine notations and converting them into this notation may cause significant extra modeling effort or may lead to loss of important information required for assumption learning.

In this paper, we propose EPIcuRus (assumPtIon geneRation approach for CPS). EPIcuRus is tailored for the analysis of Simulink models, which are commonly used in early stages of development for cyber-physical systems. EPIcuRus receives as input a software component M and a requirement ϕ such that M violates ϕ for some (but not all) of its inputs. It automatically infers a set of conditions (i.e., an environment assumption) on the inputs of M such that M satisfies ϕ when its inputs are restricted by those conditions. EPIcuRus combines machine learning and search-based testing to generate an environment assumption. Search-based testing is used to automatically generate a set of test cases for M exercising requirement ϕ such that some test cases are passing and some are failing. The generated test cases and their results are then fed into a machine learning decision tree algorithm to automatically infer an assumption A on the inputs of M such that M is likely to satisfy ϕ when its inputs are restricted by A. Model checking is used to validate and environment assumption A by checking if M guarantees ϕ when it is fed with inputs satisfying A. If not validated, EPIcuRus continues iteratively until it finds assumptions that can be validated by model checking or runs out of it search time budget. To increase the efficiency and effectiveness of EPIcuRus we design a novel test generation technique, namely Important Features Boundary Test (IFBT). IFBT guides the test generation by focusing on input features with highest impact on the requirement satisfaction and the areas of the search space where test cases change from passing to failing. At each iteration, EPIcuRus uses the decision tree from the previous iteration to obtain this information.

We evaluated EPIcuRus using four industrial models with 18 requirements. Our evaluation aims to answer two questions: (i) If IFBT, our proposed test generation policy, can outperform existing test generation policies proposed in the literature (uniform random (UR) and adaptive random testing (ART)) in learning assumptions more effectively and efficiently (**RQ1**), and (ii) if EPIcuRus, when used with its optimal test generation policy, is effective and efficient for practical usages (**RQ2**). Our results show that (1) for all the 18 requirements, EPIcuRus is able to compute an assumption ensuring the satisfaction of that requirement, and further, EPIcuRus generates $\approx$ 78% of these assumptions in less than one hour, and (2) IFBT outperforms UR and ART by generating $\approx$ 13% more valid assumptions while requiring $\approx$ 65% less time.

The contributions of this work are summarized in the following:

- We present the EPIcuRus assumption generation approach and provide a concrete and detailed implementation of EPIcuRus (Sections 3 and 5).
- We formulate the assumption generation problem for Simulink models (Section 4).
- We describe how we infer constraints from decision trees and how the constraints can be translated into logic-based assumptions over signal variables such that they can be analyzed by an industrial model checker, namely QVtrace [6] (Section 5.3).
- We introduce IFBT, a novel test case generation technique, that aims at increasing the efficiency and effectiveness of EPIcuRus (Section 5.4).
- We evaluate EPIcuRus on four real industrial models and 18 requirements.

Structure. Section 2 introduces the Autopilot running example. Section 3 outlines EPIcuRus and its pre-requisites. Section 4 formalizes the assumption generation problem. Section 5 presents how EPIcuRus is developed. Section 6 evaluates EPIcuRus, and Section 7 discusses the threats to validity. Section 8 compares with the related work and Section 9 concludes the paper.

2 CONTEXT AND RUNNING EXAMPLE

In this section, we motivate our approach using a running example and describe the pre-requisites for EPIcuRus, our automated assumption generation framework. Our running example, which we refer to it as Autopilot, is a software component in the autopilot system [2] of a De Havilland Beaver [3] aircraft. Autopilot issues commands to the plane actuators to control the aircraft's orientation (Pitch, Roll, and Yaw angles). The Autopilot component receives its inputs from two other components: a route planner component that computes the aircraft route, and a flight director component (a.k.a. auto-throttle) which provides Autopilot, among other inputs, with the throttle force required to adjust the aircraft speed. For the De Havilland Beaver aircrafts, the throttle input of Autopilot, which is required to help the aircraft reach its desired altitude, is typically provided by the pilot as such aircrafts may not contain a flight director component.

The Autopilot model is specified in the Simulink language [21]. The Simulink model of Autopilot is expected to satisfy a number of requirements, one of which is given below:

ϕ_1 ::= *When the autopilot is enabled, the aircraft altitude should reach the desired altitude within 500 seconds in calm air.*

The above requirement ensures that Autopilot controls the aircraft such that it reaches the input desired attitude within a given time limit (i.e., 500 sec in this requirement). To determine whether, or not, Autopilot satisfies the requirement ϕ_1, we convert the requirement into a formal property and use QVTrace [6], a commercial SMT-based model checker for Simulink models. QVTrace, however, fails to demonstrate that the Autopilot Simulink model satisfies the requirement ϕ. Neither can QVTrace show that Autopilot satisfies $\neg\phi$, indicating that for some inputs, Autopilot violates ϕ, and for some, it satisfies ϕ. Note that if the model satisfies either ϕ or $\neg\phi$, there is no need for generating an input assumption.

One of the reasons that Autopilot does not satisfy ϕ_1 is that Autopilot is expected to operate under the following environmental assumption [8]:

α_1 ::= *To provide the aircraft with enough boost so that it can reach the desired altitude, the pilot should manually adjust the power given to the engines of the aircraft to ensure that the aircraft does not enter a stall condition.*

In other words, Autopilot can ensure requirement ϕ_1 only if its throttle boost input satisfies the α_1 assumption. For example, if the pilot does not provide Autopilot with sufficient throttle force when the aircraft is in climbing mode, the aircraft will not reach its desired altitude and Autopilot will fail to satisfy ϕ_1.

Objective. Without including assumption α_1, in the above example, we may falsely conclude that Autopilot is faulty as it does not satisfy ϕ_1. However, after restricting the inputs of Autopilot with an appropriate assumption, we can show that Autopilot satisfies ϕ_1. Hence, there is no fault in the internal algorithm of Autopilot. In this paper, we provide EPIcuRus, *an automated approach to infer environment assumptions for system components such that they, after being constrained by the assumptions, can satisfy their requirements.* EPIcuRus is applicable under the following pre-requisites (or contextual factors):

Prerequisite-1. *The component M to be analyzed is specified in the Simulink language.* Simulink [21] is a well-known and widely-used language for specifying the behavior of cyber-physical systems such as those used in the automotive and aerospace domains.

Prerequisite-2. *The requirement ϕ the component has to satisfy is specified in a logical language.* This is to ensure that the requirements under analysis can be evaluated by model checkers or converted into fitness functions required by search-based testing. Both model checking and search-based testing are parts of EPIcuRus.

Prerequisite-3. *The satisfaction of the requirements of interest over the considered component can be verified using a model checker.*

Prerequisite-4. *The model satisfies neither the requirement nor its negation since, otherwise, an input assumption is not needed.*

3 EPICURUS OVERVIEW

Fig. 1 shows an overview of EPIcuRus which is described by Algorithm 1. EPIcuRus iteratively performs the following three main steps: (1) Test generation where a set *TS* of test cases that exercise

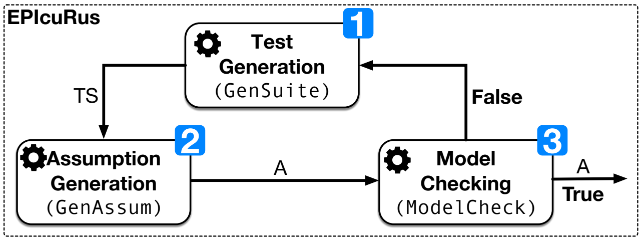

EPIcuRus
1
Test Generation (GenSuite)
TS
False
2
Assumption Generation (GenAssum)
A
3
Model Checking (ModelCheck)
A
True

Figure 1: An overview on the EPIcuRus framework.

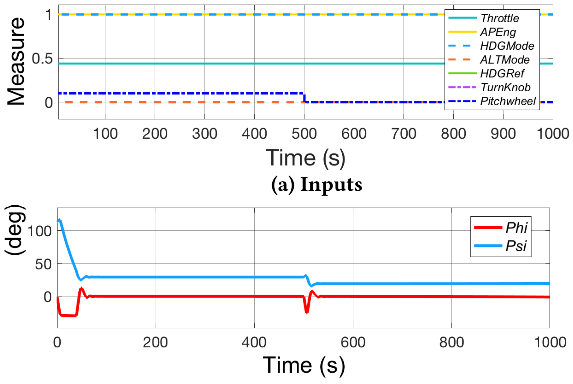
Algorithm 1 The EPIcuRus Approach.

```
Inputs. $\mathcal{M}$: The Simulink Model
        $\phi$: Test Requirement of Interest
        opt: Options
        MAX_IT: Max Number of iterations
Outputs. $A$: The assumption
function A=EPIcuRus($\mathcal{M}$, $\phi$, opt, MAX_IT)
    Counter=0; TS= [];                        ▷ Variables Initialization
    do
        TS=GenSuite($\mathcal{M}$, $\phi$, TS, opt)   ▷ Test Suite Generation
        A=GenAssum(TS);                       ▷ Assumption Generation
        Counter++;                            ▷ Increases the counter
    while not ModelCheck(A, $\mathcal{M}$, $\phi$) and Counter<MAX_IT
    return $A$;
end function
```

M with respect to requirement ϕ is generated. The test suite TS is generated such that it includes both passing test cases (i.e., satisfying ϕ) and failing test cases (i.e., violating ϕ); (2) Assumption generation where, using the test suite TS, an assumption A is generated such that M restricted by A is likely to satisfy ϕ; (3) Model checking where M restricted by A is model checked against ϕ. We use the notation $\langle A \rangle M \langle \phi \rangle$ (borrowed from the compositional reasoning literature [24]) to indicate that M restricted by A satisfies ϕ. If our model checker can assert $\langle A \rangle M \langle \phi \rangle$, an assumption is found. Otherwise, we iterate the EPIcuRus loop. The approach stops when an assumption is found or a set time budget elapses.

Simulink Models. Simulink [7] is a data-flow-based visual language for model-based design. Each Simulink model has a number of inputs and a number of outputs. We denote a test input for M as $\bar{u} = \{u_1, u_2 \ldots u_m\}$ where each u_i is a signal for an input of M, and a test output for M as $\bar{y} = \{y_1, y_2 \ldots y_n\}$ where each y_i is a signal for some output of M. Simulink models can be executed using a simulation engine that receives a model M and a test input $\bar{u}$ consisting of signals over a time domain $\mathbb{T}$, and computes the test output $\bar{y}$ consisting of signals over the same time domain $\mathbb{T}$. A *time domain* $\mathbb{T} = [0, b]$ is a non-singular bounded interval of $\mathbb{R}$. A *signal* is a function $f : \mathbb{T} \rightarrow \mathbb{R}$. A *simulation*, denoted by $H(\bar{u}, M) = \bar{y}$, receives a test input $\bar{u}$ and produces a test output $\bar{y}$.

For example, Fig. 2a shows a test input for the autopilot example, where signals are defined over the time domain $[0, 10^3]$, and Fig. 2b shows the corresponding test output. The Simulink simulation engine takes around 30s to simulate the behavior of the system over the time domain $[0, 10^3]$.[1]



(a) Inputs

(b) Outputs

Figure 2: An example of Input/Output signals of the AutopilotSimulink model.

Test generation (1). The goal of the test generation step is to generate a test suite TS of test cases for M such that some test inputs lead to the violation of ϕ and some lead to the satisfaction of ϕ. Without a diverse test suite TS containing a significant proportion of passing and failing test cases, the learning algorithm used in the assumption generation step is not able to accurately infer an assumption. We use search-based testing techniques [12, 23, 59] for test generation and rely on simulations to run the test cases. Search-based testing allows us to guide the generation of test cases in very large search spaces. It further provides the flexibility to tune and guide the generation of test inputs based on the needs of our learning algorithm. For example, we can use an explorative search strategy if we want to sample test inputs uniformly or we can use an exploitative strategy if our goal is to generate more test inputs in certain areas of the search space. For each generated test input, the underlying Simulink model is executed to compute the output. The verdict of the property of interest (ϕ) is then evaluated based on the simulation. Note that, while inputs that satisfy and violate the property of interest can also be extracted using model checkers, due to the large amount of data needed by ML to derive accurate assumptions, we rely on simulation-based testing. Further, it is usually faster to simulate models rather than to model check them. Hence, simulation-based testing leads to the generation of larger amounts of data within a given time budget compared to using model checkers for data generation.

Assumption generation (2). Given a requirement ϕ and a test suite TS generated by the test generation step, the goal of the assumption generation step is to infer an assumption A such that M restricted based on A is likely to satisfy ϕ. The test inputs generated by the test generation step are labelled by the verdict value (pass or fail). We use Decision Tree (DT) learners to derive an assumption based on test inputs labelled by binary verdict values. DT are supervised learning techniques that are trained based on labeled data and can address regression or classification problems. In this paper, we rely on classification trees to represent assumptions since our test cases are labelled by binary pass/fail values.

Fig. 3 shows an example of a classification DT used to learn an assumption based on labelled test cases for `Autopilot`. The internal nodes of the tree (a.k.a. split nodes) are associated with

[1] Machine 4-Core i7.2.5 GHz.16 GB of RAM

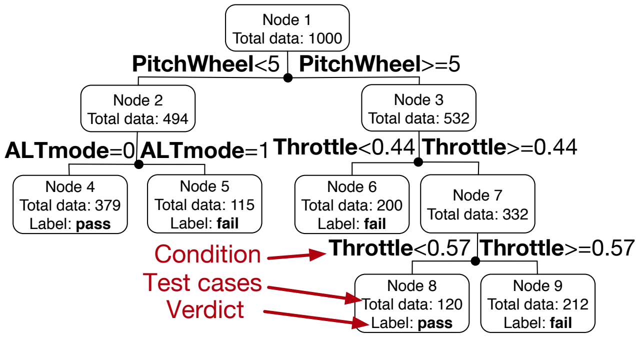


Figure 3: An example of classification tree constraining some of the inputs of the Autopilot running example.

conditions in the form $a \sim v$ where a is an attribute, v is a value and $\sim \in \{<, \geq, =\}$. Each leaf node is labelled with pass or fail depending on the label of the majority of instances that fall in that leaf node. The DT algorithm recursively identifies attributes (elements in the training set) to be associated with each internal node, defines conditions for these attributes and branches the training instances according to the values of the selected attribute. Each branching point corresponds to a binary decision criterion expressed as a predicate. Ideally, the algorithm terminates when all the leaf nodes are pure, that is when they contain instances that all have the same classification.

Model checking (3). This step checks whether the assumption A generated by the assumption generation step is accurate. Note that the DT learning technique used in the assumption generation step, being a non-exhaustive learning algorithm, cannot ensure that A guarantees the satisfaction of ϕ for M. Hence, in this step, we use a model checker for Simulink models to check whether M restricted by A satisfies ϕ, i.e., whether $\langle A \rangle M \langle \phi \rangle$ holds. We use QVTrace [6] to exhaustively check the assumption A generated in the assumption generation step. QVTrace takes as input a Simulink model and a requirement specified in QCT which is a logic language based on a fragment of first-order logic. In addition, QVtrace allows users to specify assumptions using QCT, and to verify whether a given requirement is satisfied for all the possible inputs that satisfy those assumptions. QVtrace uses SMT-based model checking (specifically Z3 BMC [28]) to verify Simulink models. The QVTrace output can be one of the following: (1) *No violation exists* indicating that the assumption is valid (i.e., $\langle A \rangle M \langle \phi \rangle$ holds); (2) *No violation exists for* $0 \leq k \leq k_{max}$. The model satisfies the given requirement and assumption in the time interval $[0, k_{max}]$. However, there is no guarantee that a violation does not occur after k_{max}; (3) *Violations found* indicating that the assumption A does not hold on the model M; and (4) *Inconclusive* indicating that QVTrace is not able to check the validity of A due to scalability and incompatibility issues.

4 ASSUMPTION GENERATION PROBLEM

In this section, we formulate the assumption generation problem for Simulink models. Let M be a Simulink model. An assumption A for M constrains the inputs of M. Each assumption A is the disjunction $C_1 \vee C_2 \vee \ldots \vee C_n$ of one or more constraint $C = \{C_1, C_2, \ldots, C_n\}$. Each constraint in C is a first-order formula in the following form:

$$\forall t \in [\tau_1, \tau_1'] : P_1(t) \wedge \forall t \in [\tau_2, \tau_2'] : P_2(t) \wedge \ldots \wedge \forall t \in [\tau_n, \tau_n'] : P_n(t)$$

where each $P_i(t)$ is a predicate over the model input variables, and each $[\tau_i, \tau_i'] \subseteq \mathbb{T}$ is a time domain. Recall from Section 3 that $\mathbb{T} = [0, b]$ is the time domain used to simulate M. An example constraint for the `Autopilot` model discussed in Section 2 is the constraint C_1 defined as follows:

$$\forall t \in [0, T] : (ALTMode(t) = 0) \wedge (0.4 \leq Throttle(t) < 0.5)$$

The constraint C_1 contains two predicates that respectively constrain the values of the input signals $ALTMode$ and $Throttle$ of the `Autopilot` model over the time domain $[0, T]$.

Let $\bar{u}$ be a test input for a Simulink model M, and let C be a constraint over the inputs of M. We write $\bar{u} \models C$ to indicate that the input $\bar{u}$ satisfies the constraint C. For example, the input $\bar{u}$ for the `Autopilot` model and described using the diagrams in Fig. 2 satisfies the constraint C_1. Note that for Simulink models, test inputs are described as functions over a time domain $\mathbb{T}$, and similarly, we define constraints C as a conjunction of predicates over the same time domain or its subsets.

Let $A = C_1 \vee C_2 \vee \ldots \vee C_n$ be an assumption for model M, and let $\bar{u}$ be a test input for M. The input u satisfies the assumption A if $\bar{u} \models A$. For example, consider the assumption $A = C_1 \vee C_2$ where

$$C_1 ::= \forall t \in [0, T] : (ALTMode(t) = 0) \wedge (HDG_{Ref}(t) < 90)$$
$$C_2 ::= \forall t \in [0, T] : (ALTMode(t) = 0) \wedge (HDG_{Ref}(t) < 20)$$

The input $\bar{u}$ in Fig. 2 satisfies the assumption A since it satisfies the constraint C_1.

Let A be an assumption, and let $\mathcal{U}$ be the set of all possible test inputs of M. We say $U \subseteq \mathcal{U}$ is a *valid input set* of M restricted by the assumption A if for every input $\bar{u} \in \mathcal{U}$, we have $\bar{u} \models A$. Let ϕ be a requirement for M that we intend to verify. Recall from Section 3 that our search-based testing framework works based on a fitness function $\textsc{Fn}$ that is defined to derive test inputs violating a given requirements ϕ. For every test input $\bar{u}$ and its corresponding test output $\bar{y}$, our fitness function $\textsc{Fn}(\bar{u}, \bar{y}, \phi)$ returns a value that determines the degree of violation or satisfaction of ϕ when M is executed for test input $\bar{u}$. Specifically, following existing research on search-based testing of Simulink models [11, 48, 49] we define the fitness function $\textsc{Fn}$ as a function that takes a value between $[-1, 1]$ such that a negative value indicates that the test inputs $\bar{u}$ reveals a violation of ϕ and a positive or zero value implies that the test input $\bar{u}$ is passing (i.e., does not show any violation of ϕ). The fitness function $\textsc{Fn}$ allows us to distinguish between different degrees of satisfaction and failure and hence guide the generation of test inputs for Simulink models within our search-based testing framework. When $\textsc{Fn}$ is positive, a closer value to 0 indicates that we are close to violating, although not yet violating, requirement ϕ, while a value close to 1 shows that the model is far from violating ϕ. Dually, when $\textsc{Fn}$ is negative, a fitness value close to 0 shows a less severe failure than a value close to -1.

Definition 1. *Let A be an assumption, let ϕ be a requirement for M, and let $\textsc{Fn}$ be the fitness function defined to assess requirement ϕ. We say the degree of satisfaction of the requirement ϕ over model M restricted by the assumption A is v , i.e., $\langle A \rangle M \langle \phi \rangle = v$, if*

$$v = \underset{\bar{u} \in U}{min} (\textsc{Fn}(\bar{u}, \bar{y}, \phi))$$

where $\bar{y}$ is the test output for any test input $\bar{u} \in U$.

We say an assumption A is v-*safe* for a model M and its requirement ϕ, if $\langle A \rangle M \langle \phi \rangle > v$. As discussed earlier, we define the fitness function such that a fitness value v larger than or equal to 0 indicates that the requirement under analysis is satisfied. Hence, when an assumption A is 0-*safe*, the model M restricted by A satisfies ϕ.

For a given model M, a requirement ϕ and a given fitness value v, we may have several assumptions that are v-*safe*. We are typically interested in identifying the weakest v-safe assumption since the weakest assumption leads to the largest valid input set U, and hence is less constraining. Let A_1 and A_2 be two different v-*safe* assumptions for a model M and its requirement ϕ. We say A_1 is more *informative* if it is weaker than A_2, i.e., $A_1 \Rightarrow A_2$. In this paper, provided with a model M, a requirement ϕ and a desired fitness value v, our goal is to generate the weakest (most informative) v-safe assumption. We note that our approach, while guaranteeing the generation of v-safe assumptions, does not guarantee that the generated assumptions are the most informative. Instead, we propose heuristics to maximize the possibility of the generation of the most informative assumptions and evaluate our heuristics empirically in Section 6.

5 IMPLEMENTATION

In the following, we describe the implementation of each step of Algorithm 1 (i.e., the main loop of EPIcuRus). Since our implementation relies on QVTrace, which can not handle quantitative fitness values, we are considering 0-safe assumptions. Section 5.1 describes alternative test case generation procedures for line 4 of Algorithm 1 (1). Section 5.2 presents our assumption generation procedure for line 5 of Algorithm 1 (2). Section 5.3 describes the model checking procedure for line 7 of Algorithm 1 (3). Section 5.4 presents our test case generation procedure (IFBT) for line 4 of Algorithm 1 (1).

5.1 Test Generation

Algorithm 2 shows our approach to generate a test suite i.e., a set of test inputs together with their fitness values. Note that EPIcuRus iteratively builds a test suite, and at each iteration, it extends the test suite generated in the previous iteration. To do so, Line 2 copies the old test suite into the new test suite. Then, within a for-loop, the algorithm generates one test input (Line 4) in each iteration and executes the test input to compute its fitness value (Line 5). The new test suite is finally returned (Line 8). Below, we describe our test generation strategies as well as our fitness functions.

Definition of the Fitness Function — Fn. We use existing techniques [32, 49] on translating requirements into quantitative fitness functions to develop Fn corresponding to each requirement ϕ. Fitness functions generated by these techniques serve as distance functions, estimating how far a test is from violating ϕ, and hence, they can be used to guide the generation of test cases. In addition, we can infer from such fitness functions whether, or not, a given requirement ϕ is satisfied or violated by a given test case. Specifically, if $\text{Fn}(\bar{\mathrm{u}}, \bar{\mathrm{y}}, \phi) \geq 0$, the output $\bar{\mathrm{y}}$ generated by the test input $\bar{\mathrm{u}} = \{u_1, u_2 \dots u_m\}$ satisfies ϕ, and otherwise, $\bar{\mathrm{y}}$ and $\bar{\mathrm{u}}$ violate ϕ.

Specifying Test Cases. Algorithm 2 iteratively generates a new test input $\bar{\mathrm{u}}_i$ for the model $\mathcal{M}$. Since $\mathcal{M}$ is a Simulink model, the algorithm should address the following test generation challenges:

- *It should generate inputs that are signals (functions over time) instead of single values since Simulink model inputs are signals.*
- *Input signals should be meaningful for the model under analysis and should be such that they can be realistically generated in practice.* For example, a signal representing an airplane throttle command is typically represented as a sequence of step functions and not as a high frequency sinusoidal signal.

Algorithm 2 Test Suite Generation.

```
Inputs. $\mathcal{M}$: The Simulink Model
        $\phi$: The property of interest
        TSOld: Old Test Suite
        opt: Options
Outputs. TSNew: New Test Suite
1: function TSNew=GenSuite($\mathcal{M}$, $\phi$, TSOld, opt)
2:   TSNew=TSOld;                              ▷ Test Suite Initialization
3:   for i=0; i<opt.TestSuiteSize; i++
4:     $\bar{\mathrm{u}}_i$=GenTest($\mathcal{M}$, TSOld, opt);        ▷ Test Case Generation
5:     $v_i$=Fn($\bar{\mathrm{u}}_i$, $\mathcal{M}(\bar{\mathrm{u}}_i)$);              ▷ Execute of the Test Case
6:     TSNew=TSNew∪{$\langle \bar{\mathrm{u}}_u, v_i \rangle$}  ▷ Add the Test to the Test Suite
7:   end for
8:   return TSNew
9: end function
```

To generate signals, we use the approach of Matlab [9, 25] that encodes signals using some parameters. Specifically, each signal u_u in $\bar{\mathrm{u}}$ is captured by $\langle int_u, R_u, n_u \rangle$, where int_u is the interpolation function, $R_u \subseteq \mathbb{R}$ is the input domain, and n_u is the number of control points. Provided with the values for these three parameters, we generate a signal over time domain $\mathbb{T}$ as follows: (i) we generate n_u control points equally distributed over the time domain $\mathbb{T}$, i.e., positioned at a fixed time distance I; (ii) we assign randomly generated values $c_{u,1}, c_{u,2}, \dots, c_{u,n_u}$ within the domain R_u to each control point; and (iii) we use the interpolation function int_u to generate a signal that connects the control points. The interpolation functions provided by Matlab includes among others, linear, piecewise constant and piecewise cubic interpolations, but the user can also define custom interpolation functions. To generate realistic inputs, the engineer should select an appropriate value for the number of control points (n_u) and choose an interpolation function that describes with a reasonable accuracy the overall shape of the input signals for the model under analysis. Based on these inputs, the test generation procedure has to select which values $c_{u,1}$, $c_{u,2}$, $\dots$, c_{u,n_u} to assign to the control points for each input u_u.

For example, the signals in Fig. 2a have three control points respectively representing the values of the signals at time instants 0, 500, and 1000. The signals AP_{Eng}, $HDGMode$, $AltMode$ and $Throttle$, HDG_{Ref}, $TurnKnob$, $Pitchwheel$ are respectively generated using boolean and piecewise constant interpolations.

Generating Test Cases. The test suite generation technique uses a test case generation policy p to select values for control points related to each test input. Some test case generation policies, such as Adaptive Random Testing (ART) [12, 22, 23] and Uniform Random Testing (UR) [12, 13] can be used to generate a diverse set of test inputs that are evenly distributed across the search space. UR samples each control point value following a random uniform distribution, and ART randomly samples values from the search space by maximizing the distance between newly selected values and the previously generated ones, hence increasing the distance between the sampled values and ensuring that they are evenly spread over the search space.

Execution of the Test Cases. Our procedure uses the Simulink simulator (see Section 2) to generate output signals $\overline{\mathrm{y}} = \mathcal{M}(\overline{\mathrm{u}})$ associated with the generated test input $\overline{\mathrm{u}}$. Using the fitness function Fn, we obtain the verdict (pass/fail) value for each test input $\overline{\mathrm{u}}$ and output $\overline{\mathrm{y}}$ of the model under analysis. For example, the Simulink simulator generates the output in Fig. 2b from the input in Fig. 2a. The fitness value for this input and output computed based on the requirement ϕ_1, described in Section 2, is ≈ 0.72.

5.2 Assumption Generation

We use machine learning to automatically compute assumptions. Specifically, the function GenAssum (see Algorithm 1) infers an assumption by learning patterns from the test suite data (TS). This is done by (i) learning a classification tree that predicts requirement satisfaction; (ii) extracting from the tree a set of predicates defined over the control points of the input signals of the model under analysis; and (iii) transforming the predicates into constraints over input signals, as defined in Section 4, such that they can be fed as an assumption into QVTrace. The steps (i), (ii), and (iii), which are fully integrated in the Matlab framework, are described as follows.

Learning Classification Trees. We use classification trees to mine an assumption from sets of test inputs labelled with pass/fail verdict. Classification trees are a widely adopted technique to identify interpretable patterns from data [52]. Recall from Section 5.1 that test inputs are captured in terms of value assignments to signal control points. Hence, classification trees learn conditions on the values of signal control points. More specifically, we use the function `fitctree` of Matlab [5] to build the trees. This function implements the standard CART algorithm [19]. EPIcuRus enables the users to select parameter values to configure the CART algorithm implemented by the `fitctree` function [5]. The user can select the values of these parameters by considering the characteristics of the model under analysis and some trial and error experiments. Fig. 3 reports an example of decision tree computed by the assumption generation procedure for the autopilot example, when each model input is encoded using a single control point.

We follow a standard procedure to extract conditions on control points from a classification tree [63]: Each pure leaf labeled "pass" (i.e., each leaf containing 100% test inputs satisfying ϕ) yields a conjunctive predicate which is obtained by conjoining all the conditions on the path from the tree root to the leaf (see below):

$$c_{u,1} \sim v_1 \wedge \ldots \wedge c_{u,q} \sim v_k \wedge \ldots \wedge c_{z,j} \sim v_h$$

where $c_{x,y}$ denotes the yth control point associated with input x, and each condition $c_{x,y} \sim v$ such that $\sim \in \{<, \geq, =\}$ and $v \in \mathbb{R}$ is a constraint over the control point $c_{x,y}$. For example, in the above conjunction, the control points $c_{u,1}$ and $c_{u,q}$ are related to input u and the control point $c_{z,j}$ is related to input z.

From conjunctions of predicates to a QVtrace assumption. The conjunctive predicates constrain control points. Before, we can use them with our model checker QVTrace, they have to be converted into constraints over signal variables as defined in Section 3. To do so, we use the rules provided in Table 1 to convert each predicate over control points, into constraints over signal variables. Specifically, the rules in Table 1 are as follows:

- *When the conjunction includes $c_{u,j} \sim v \wedge c_{u,j+1} \sim v'$, the predicate $c_{u,j} \sim v$ is replaced by a constraint over input signal u using the cases 1, 2, 3, and 4 rules in Table 1 depending on the type of the relation $\sim$.* Note that in this case, the conjunction includes predicates over two adjacent control points related to the same input signal (i.e., $c_{u,j}$ and $c_{u,j+1}$ are two consecutive control points related to the input signal u). For brevity, in Table 1, we present only the cases for $<$ and $\geq$ and when the input u is a real signal. The cases in which u is a boolean signal and $\sim_1$ and $\sim_2$ are "=" are similar to the one reported in Table 1 and are presented in our online appendix [4].

Intuitively, the conversion rules in Table 1 assume that the consecutive control points are connected by a linear interpolation function. While other functions can also be considered, and custom functions can be defined by users, we believe that linear interpolation functions provide a good compromise between simplicity and realism. In our evaluation (see Section 6), we assess the impact of our assumption on the effectiveness of our approach. Note that we only assume that consecutive control points are connected by a linear function to be able to generate an assumption over input signals. Otherwise, input signals can be encoded using various (non-linear) interpolation functions and our test generation step is able to generate input signals using any given interpolation function.

- *When the conjunction includes $c_{u,j} \sim v$, but no control point adjacent to $c_{u,j}$ appears in the conjunction, then the predicate $c_{u,j} \sim v$ is replaced by a constraint over input signal u using the cases 5 and 6 rules in Table 1 depending on the type of the relation $\sim$.* In this case, we assume that the resulting constraint holds from the time instant t_1 associated with the control point $c_{u,j}$ to the time instant $t_1 + \frac{I}{2}$ where I is the time step between two consecutive control points.

Following the above rules, we convert any conjunction of predicates over control points into a constraint C defined over input signals. Note that provided with a classification tree, we obtain a conjunction of predicates for every pure leaf labelled with a pass verdict. The set of all conjunctions is then converted into an assumption $A = C_1 \vee C_2 \vee \ldots \vee C_n$ where each C_i is obtained by translating one conjunction of predicates using the rules in Table 1.

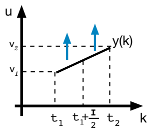
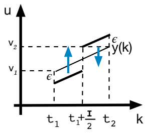
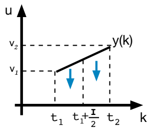
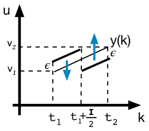
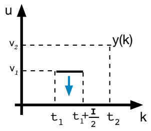
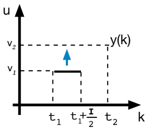
5.3 Checking Accuracy

Checking accuracy aims at verifying whether a given assumption is $v-$safe. To do so, we rely on QVtrace which exhaustively verifies a given model constrained with some assumption against a formal requirement expressed in the QCT language. As discussed in Section 3, QVTrace generates four kinds of outputs. When it returns "No violation exists", or "No violation exists for $0 \leq \mathrm{k} \leq \mathrm{k}_{max}$", we conclude that the given assumption A is v-safe, i.e., the model under analysis when constrained by A satisfies the given formal requirement. Otherwise, we conclude that the assumption is not v-safe and has to be refined or modified.

5.4 IFBT — Important Features Boundary Test

Learning v-safe assumptions in our context requires a sufficiently large test suite, which is necessarily generated by simulating the

Table 1: Generating the predicates of the constraint.

	Condition		QCT Clause	Condition		QCT Clause
	Case 1			Case 2		
Two consecutive control points	$c_{u,j} \geq v_1$ $c_{u,j+1} \geq v_2$		$\forall t \in [t_1, t_1 + \frac{I}{2}]: u(t) \geq y(t)$ $\wedge$ $\forall t \in [t_1 + \frac{I}{2}, t_2]: u(t) \geq y(t)$	$c_{u,j} \geq v_1$ $c_{u,j+1} < v_2$		$\forall t \in [t_1, t_1 + \frac{I}{2}]: u(t) \geq y(t) - \epsilon$ $\wedge$ $\forall t \in [t_1 + \frac{I}{2}, t_2]: u(t) < y(t) + \epsilon$
	Case 3			Case 4		
	$c_{u,j} < v_1$ $c_{u,j+1} < v_2$		$\forall t \in [t_1, t_1 + \frac{I}{2}]: u(t) < y(t)$ $\wedge$ $\forall t \in [t_1 + \frac{I}{2}, t_2]: u(t) < y(t)$	$c_{u,j} < v_1$ $c_{u,j+1} \geq v_2$		$\forall t \in [t_1, t_1 + \frac{I}{2}]: u(t) < y(t) + \epsilon$ $\wedge$ $\forall t \in [t_1 + \frac{I}{2}, t_2]: u(t) \geq y(t) - \epsilon$
	Case 5			Case 6		
One control point	$c_{u,j} < v_1$		$\forall t \in [t_1, t_1 + \frac{I}{2}]: u(t) < v_1$	$c_{u,j} \geq v_1$		$\forall t \in [t_1, t_1 + \frac{I}{2}]: u(t) \geq v_1$

* $t_1 = I \cdot j$ and $t_2 = I \cdot (j+1)$ and $y(t) = \frac{v_2 - v_1}{I} \cdot (t - t_1) + v_1$ and $I = t_2 - t_1$: time distance.

model under analysis a high number of times. To reduce the number of test cases that have to be generated for assumption learning, we propose a new test case generation strategy, namely Important Features Boundary Test (IFBT). The idea is to generate test cases in areas of the input domain that are more informative for the ML procedure used to learn v-safe assumptions. We make the following conjectures:

Conj 1: The values assigned to *some* control points have a higher impact on the fitness value than the values assigned to others. Identifying such control points and focusing the search on them enables more effective learning of v-safe assumptions.

Conj 2: Generating test cases in boundary areas of the input domain where the fitness value changes from being lower than v to being greater than v enables more effective learning of v-safe assumptions.

Based on the above intuitive conjectures, IFBT generates a test suite by following the steps of Algorithm 3 detailed below.

Step 1 (Line 3): We build a regression tree from the previously generated test suite TSOld that describes the relationship between the values assigned to the control points (i.e., the features of the regression tree) and the fitness value. We choose to use regression trees since we want to learn the impact of control point values on the continuous fitness value and this is therefore a regression problem. The leaves of the tree contain test cases that are characterized by similar fitness values for control points.

Step 2 (Line 4): This step follows from conjecture Conj 2 and attempts to generate boundary test cases. To do so, among all leaves in the tree, we pick the two leaves with average fitness values that are the closest to the selected v threshold–that is 0 in our case as described above, one being below the threshold and the other one above. We extract the test cases TC associated with these nodes. These test cases are the ones closest to boundary where the fitness value changes from being greater than v to being lower than v and are therefore used to generate the new test cases in the following steps of the algorithm.

Algorithm 3 IFBT - Important Features Boundary Test.

Inputs. $\mathcal{M}$: The Simulink Model
ϕ: The property of interest
TSOld: Old Test Suite
opt: Options
Outputs. TSNew: New Test Suite

```
1:  function TSNew=GenSuite($\mathcal{M}$, $\phi$, TSOld, opt)
2:    TSNew=TSOld
3:    RT=GenRegressionTree(TSOld);  ▷ Learn a Regression Tree
4:    TC=GetTests(RT,v);                 ▷ Get Tests on Leaves
5:    ⟨Feat,Rng⟩=GetImpF(RT,opt.num)          ▷ Get features
6:    for i=0; i<opt.TestSuiteSize; i++
7:      $\bar{u}_i$ = GenTest(TC.next,Feat,Rng,opt) ▷ Test Case Generation
8:      $v_i$=Fn($\bar{u}_i$, $\mathcal{M}(\bar{u}_i)$);        ▷ Execute of the Test Case
9:      TSNew=TSNew∪{⟨$\bar{u}_u$, $v_i$⟩}  ▷ Add the Test to the Test Suite
10:   endfor
11:   return TSNew
12: end function
```

Step 3 (Line 5): We save in the variable Feat the subset of the *opt.num* most important features of the regression tree. This step follows from conjecture Conj 1 as feature importance captures how much the value assigned to the feature (control point) influences the fitness value. Furthermore, for every control point $c_{u,j}$ that belongs to the set of features Feat, it computes a range (saved in Rng) associated with the control point based on the conditions that constrain the control point $c_{u,j}$ in the regression tree RT. For every condition $c_{u,j} \sim v_1$ that constrains control point $c_{u,j}$ in RT, the interval $[v_1 - opt.perc \cdot v_1, v_1 + opt.perc \cdot v_1]$ is added to the range Rng associated with the control point $c_{u,j}$. For example, if a

Table 2: Identifier, name, description, number of blocks of the Simulink model (#Blocks), number of inputs (#Inputs) and number of requirements (#Reqs) of our study subjects.

ID	Name	Description	#Blocks	#Inputs	#Reqs
TU	Tustin	A numeric model that computes integral over time.	57	5	2
REG	Regulator	A typical PID controller.	308	12	6
TT	Two Tanks	A two tanks system where a controller regulates the incoming and outgoing flows of the tanks.	498	2	7
FSM	Finite State Machine	A finite state machine that turns on the autopilot mode in case of some environment hazard.	303	4	3

constraint $c_{u,1} < 2$ associated with control point $c_{u,1}$ is present in the regression tree RT, and $opt.perc = 10\%$, the interval $[1.8, 2.2]$ is added to the range Rng associated with $c_{u,1}$. This ranges will be used to generate test cases in the area of the input domain where the fitness changes from being lower than v to being greater than v, following from conjecture Conj 2.

Step 4 (Lines 6-10): We create a set of $opt.TestSuiteSize$ new test cases from the test cases in TC as follows. We iteratively select (in sequence) a test case TC.next in TC. A new test case is obtained from TC.next by changing the values of the control points in Feat according to their ranges in Rng using either UR or ART sampling. We use the acronyms IFBT-UR and IFBT-ART to respectively indicate the sampling strategy each alternative relies on.

6 EVALUATION

In this section, we empirically evaluate EPIcuRus by answering the following research questions:

- **Effectiveness and Efficiency — RQ1**: *Which test case generation policy among UR, ART, IFBT-UR and IFBT-ART helps learn assumptions most effectively and efficiently?* With this question, we investigate our four test generation policies (i.e., UR and ART discussed in Section 5.1, and IFBT-UR and IFBT-ART discussed in Section 5.4) and determine which policy can help compute the most v-safe assumptions that are the most informative while requiring the least amount of time.
- **Usefulness — RQ2**: *Can EPIcuRus generate assumptions for real world Simulink models within a practical time limit?* In this question, we investigate if EPIcuRus, when used with the best test generation policy identified in **RQ1**, can generate v-safe assumptions for our real world study subject models within a practical time limit.

Implementation and Data Availability. We implemented EPIcuRus as a Matlab standalone application and used QVtrace for checking the accuracy of the assumptions. The implementation, models and results are available at [4]. We plan to release it under an open-source license, currently being reviewed by our legal team.

Study Subjects. We consider eleven models and 92 requirements provided by QRA Corp, a verification tool vendor active in the aerospace, automotive, and defense sectors [45, 49, 54]. The models and the requirements have recently been used in a recent study to compare model testing and model checking [54]. Among the 92 requirements, only 18 requirements on four models could be handled by QVTrace (**Prerequisite-3**) and neither the requirements nor their negation could be proven by QVTrace (**Prerequisite-4**). Thus, we retain only four models and 18 requirements. Table 2 describes the four models, the number of blocks and the inputs of each model and the number of requirements for each model.

Experiment Design. To answer **RQ1** and **RQ2**, we perform the experiments below.

We execute EPIcuRus using the UR, ART, IFBT-UR, and IFBT-ART test case generation policies. For each requirement and study subject, we execute different experiments considering input signals with one (IP), two (IP′), and three (IP′′) control points. We set the number of new test cases considered at every iteration to 30 (opt.TestSuiteSize, see Algorithms 2 and 3). We considered a maximum number of iterations MAX_IT=30 (see Algorithm 1) as this is a common practice to compare test case generation algorithms [31]. For IFBT-UR and IFBT-ART, we set the value opt.num of the most important features to consider for test case generation as follows. For iterations 1 to 10, opt.num is set to one, meaning that only the most important feature is considered. For iterations 10 to 20, opt.num is set to two and iterations 20 to 30, opt.num is equal to the number of features of the decision tree. We do not know a priori the number of features that will be constrained by the final assumption. The above strategy to set opt.num starts by focusing on the most important features, and gradually enlarges the set of considered features if no valid assumptions are found. We set the value opt.perc, used by IFBT-UR and IFBT-ART to compute the next range to be considered, to $\frac{1}{1+2\cdot\text{Counter}}$. This value ensures that the size of the next interval to be considered is decreasing with the number of iterations (Counter). The intuition is that the more iterations are performed, the closer IFBT is to computing the v-safe assumption, and thus a more restricted interval can be considered. We repeated every experiment 50 times to account for the randomness of test case generation. We recorded whether EPIcuRus was able to compute a v-safe assumption, the computed v-safe assumption itself, and its execution time.

To compare *efficiency*, we consider the average execution time (`AVG_TIME`) of each test case generation policy across different experiments. To compare *effectiveness*, we consider (i) the percentage of experiment runs, among the 50 executed, (`V_SAFE`) in which each test case generation policy is able to compute a v-safe assumption; and (ii) how informative the assumptions learned by different test case generation policies are in relative terms. To measure the latter, we considered each assumption learned by a test case generation policy. We computed the number of times this assumption was more informative than another assumption learned with a different test case generation policy for the same model, requirement, experiment, and number of control points. We define the information index (`INF_INDEX`) of a test case generation policy as the sum, across the different assumptions learned with that policy, of the number of times the assumption is more informative than another assumption learned with a different test case generation policy. To check whether an assumption A_1 is more *informative* than A_2, we check if $A_1 \Rightarrow A_2$ is valid (i.e, if $A_1 \Rightarrow A_2$ is a tautology). This is done by checking whether $\neg(A_1 \Rightarrow A_2)$ is satisfiable. If $\neg(A_1 \Rightarrow A_2)$ is unsatisfiable, then $A_1 \Rightarrow A_2$ is a tautology. The satisfiability of $\neg(A_1 \Rightarrow A_2)$ is verified by an MITL satisfiability solver recently provided as a part of the TACK model checker [17, 47]. We set a timeout of two minutes for our satisfiability solver. If an unsat result is returned within this time limit $A_1 \Rightarrow A_2$ holds, otherwise, either $\neg(A_1 \Rightarrow A_2)$ is satisfiable, or a longer execution time is needed by the satisfiability checker.

As a complementary analysis, we repeat the experiment above, but instead of fixing the number of iterations across different EPIcuRus runs, we set a one hour time bound for each run of EPIcuRus on each requirement. We consider this to be a reasonable time for learning assumptions in practical settings. Note that we still execute IFBT-UR for a maximum of 30 iterations. However, if the maximum number of iterations was reached without finding a valid assumption, and the execution time was still less than one hour, EPIcuRus was re-executed. The motivation is to re-start Epicurus every 30 iterations so that it does not focus its search on a portion of the search space that has no chance of gathering useful information to learn v-safe assumptions due to a poor choice of initial inputs. Running all the experiments required approximately 33 days.[2]

6.1 RQ1 — Effectiveness and Efficiency

The scatter plot in Fig. 4 depicts the results for RQ1 obtained when comparing test generation strategies in terms of effectiveness and execution time, when running EPIcuRus a maximum of 30 iterations. The x-axis indicates the average execution time (`AVG_TIME`), our efficiency metric. The lower this time, the more efficient a test case generation policy. The y-axis indicates the percentage of cases (`V_SAFE`), across 50 runs for each requirement, for which each test case generation policy could compute a v-safe assumption. The higher the value, the higher the effectiveness of a test case generation policy. Each point of the scatter plot is labeled with the information index (`INF_INDEX`) associated to that policy. The higher this index, the more informative the v-safe assumptions computed with a test case generation policy.

As shown in Fig. 4, IFBT-UR is the best test case generation policy. IFBT-UR has indeed both the lowest average execution time `AVG_TIME` and the highest `V_SAFE` percentage. IFBT-UR generates $\approx 6-8\%$ more valid assumptions than UR and ART and requires $\approx$ 65% less time. It is only slightly better than IFBT-ART, thus showing that the main driving factor here is IFBT. Furthermore, IFBT-UR's information index `INF_INDEX` is higher than those of the other policies.

Regarding the impact of using IFBT, Fig. 4 also shows that the difference between UR and IFBT-UR is small in terms of V_SAFE%, though large in terms of the execution time. However, when fixing the execution time to a maximum of one hour, instead of iterations, IFBT-UR and UR identify a v-safe assumption respectively in 78% and 65% of the requirements. That is, when provided with an equal execution time budget, IFBT-UR outperforms UR by learning a v-safe assumption for $\approx$ 13% more requirements.

[2] We used the high performance computing cluster at [location redacted] with 100 Dell PowerEdge C6320 and a total of 2800 cores with 12.8 TB RAM. The parallelization reduced the experiments time to approximately five days.

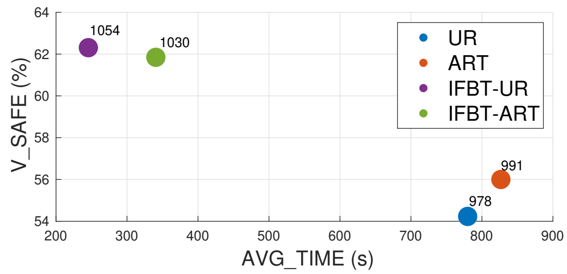


Figure 4: Comparison of the Test Case Generation Policies.

The answer to **RQ1** is that, among the four test case generation policies we compared, IFBT-UR learns the most v-safe assumptions in less time. Further, the assumptions learned by IFBT-UR are more informative than those learned by other test generation policies.

6.2 RQ2 — Usefulness

To answer RQ2, we use IFBT-UR, the best test case generation policy identified by RQ1. On average, one EPIcuRus run can compute a v-safe assumption, within one hour, for $\approx$ 78% of the requirements. Further, EPIcuRus learns assumptions that are not vacuous, and all the generated assumptions had a non-empty valid input set, i.e., none of the requirements was vacuously satisfied by the computed assumption. Across all 50 runs, which take for IFBT-UR around four hours per requirement, EPIcuRus is able to compute a v-safe assumption for all the 18 requirements of our four Simulink models. The average number of constraints in an assumption is 2.4, with 5.4 predicates on average. From that, we can conclude that the computed assumptions are relatively simple, thus suggesting they are easy to understand and that EPIcuRus does not generate much accidental complexity.

The answer to **RQ2** is that, EPIcuRus can learn non-vacuous and short assumptions for all the requirements of our subject models within reasonable time.

7 DISCUSSION AND THREATS TO VALIDITY

Our empirical evaluation confirms that our conjectures (Conj 1 and Conj 2) hold and that the effectiveness of EPIcuRus is adequate for practical usages. In the following we discuss the practical implications of EPIcuRus.

• EPIcuRus learns classification trees and converts them into classification rules. Directly learning classification rules [53] could be a better solution, as it may yield more concise assumptions. However, as classification rules are not supported by Matlab, we would have to rely on external tools, e.g., weka [63] to generate them, and further, our solution based on classification trees already works reasonably well.

• Decision trees and decision rules can only learn predicates defined over single features (i.e., single control points). That is, all the learned predicates are in the following form $c \sim v$. Hence, they are not suited to infer predicates capturing relationships among two or more features (i.e., control points). While this was not a limitation in our work, our approach can be extended to infer more complex assumptions using other machine learning techniques (e.g., more expressive rules or clustering [63]). Alternatively, we can use genetic programming [14] to learn more expressive assumptions.

• EPIcuRus generates assumptions using the rules in Table 1 that assume that consecutive control points are connected using a linear interpolation function. Our evaluation shows that this asumption provides a good compromise between simplicity and realism. The rules in Table 1 can be expanded to consider more complex interpolation functions in particular when we have specific domain knowledge about the shapes of different input signals.

• Our solution combines different techniques. Let n be the number of instances and m be the number of input features, the time complexity of the decision tree induction is $O(m \cdot n \cdot log(n)) + O(n \cdot (log(n))^2)$. The time complexity of running QVtrace is exponential in the size of the SMT instance to be solved. The time complexity of UR and ART test case generation is linear in the number of tests to be generated. For IFBT the time complexity of the decision tree induction comes in addition to the time complexity of UR and ART. As shown in our evaluation, our solution was sufficiently efficient to effectively analyze our study subjects.

Our results are subject to the following threats to validity.

External validity. The selection of the models used in the evaluation, and the features contained in those models, is a threat to external validity as it influences the extent to which our results can be generalized. However, (i) the models we considered have been previously used in the literature on testing of CPS models [49, 54], (ii) they represent realistic and representative models of CPS systems from different domains, and (iii) our results can be further generalized by additional experiments with diverse types of systems and by assessing EPIcuRus over those systems.

Internal validity. Using the same models to select the optimal test generation policy (RQ1) and to evaluate EPIcuRus (RQ2) is a potential threat to the internal validity. However, since the test generation policy is not optimized for any particular model, it is a good general compromise among many different models.

8 RELATED WORK

This section compares EPIcuRus with the following threads of research: (i) Verification, testing and monitoring CPS, (ii) Compositional and assume-guarantee reasoning for CPS, (iii) Learning assumptions for software components, and (iv) learning the values for unspecified parameters.

Verification, Testing and Monitoring of CPS. Approaches to verifying, testing and monitoring CPS have been presented in the literature (e.g., [11, 33, 46, 48, 49, 62]). These approaches, however, often assume that assumptions characterising the valid ranges of test inputs are already specified. Our work, in contrast, automatically identifies (implicit) assumptions on test inputs. Such assumptions are an important pre-requisite to ensure testing and verification results are not overly pessimistic or spurious [15, 24, 29, 36, 37].

Compositional reasoning. Assume-guarantee reasoning and design by contract frameworks have been extensively discussed in the literature (e.g., [1, 16, 26, 27, 30, 39, 50, 51]). Some recent work extends such type of reasoning to signal-based modeling formalisms such as Simulink and analog circuits (e.g., [55, 56, 61]). Our work is complementary as the assumptions learned by EPIcuRus can be used within these existing frameworks. Our work also differs from assume-guarantee testing, where assumptions defined during software design are verified during test case execution [35].

Learning Assumptions. Several approaches to automatically learn assumptions, a.k.a supervisory control problem, have been proposed (e.g., [15, 20, 24, 34, 36, 44, 57, 58]). Those approaches, however, are not applicable to signal-based formalisms (e.g., Simulink models) and are solely focused on components specified in finite-state machines. Our work bears some similarities with template-based specification mining [10, 43], however, LTL-GR(1) [10, 43] specifications used in that work are substantially different and less expressive than signal-based assumptions generated in our work. Further, we rely on testing to generate the data used to learn assumptions as, compared to model checking, testing can generate a larger amount of data in less time. One interesting idea is to combine model checking and model testing to improve the quality and performance of assumption learning. Our approach is related to a recent study [54] focused on the complementarity between model testing and model checking for fault detection purposes.

Learning Parameters. Approaches that learn (requirement) parameters from simulations have been presented in the literature [18, 40, 41]. In contrast to those approaches, our work is explicitly tailored to learning assumptions. Our technique can be considered as an extension of counterexample-guided inductive synthesis [60], where learned assumptions are exhaustively verified using an SMT-based model checker. Furthermore, our approach considers signal-based formalisms that are widely used in the CPS industry and extracts assumptions from test data. Test case generation allows us to efficiently produce a large amount of data to feed our machine learning algorithm and derive informative v-safe assumptions.

9 CONCLUSION

In this paper, we proposed EPIcuRus, an approach to automatically infer environment assumptions for software components such that they are guaranteed to satisfy their requirements under those assumptions. Our approach combines search-based software testing with machine learning decision trees to learn assumptions. In contrast to existing work where assumptions are often synthesized based on logical inference frameworks, EPIcuRus relies on empirical data generated based on testing to infer assumptions, and is hence applicable to complex signal-based modeling notations (e.g., Simulink) commonly used in cyber-physical systems.

In addition, we proposed IFBT a novel test generation technique that relies on feedback from machine learning decision trees to guide the generation of test cases by focusing on the most important features and the most informative areas in the search space. Our evaluation shows that EPIcuRus is able to infer assumptions for all the requirements in our subject studies and in 78% of the cases the assumptions are learned within just one hour. Further, IFBT outperforms simpler test generation techniques aimed at creating test input diversity, as it increases the number and the quality of the generated assumptions while requiring less time for test generation.

For future, we plan to extend our work and eliminate some of the limitations of EPIcuRus based on the ideas summarized in Section 7.

REFERENCES

[1] 2012. Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems*. *European Journal of Control* 18, 3, 217 – 238. https://doi.org/10.3166/ejc.18.217-238

[2] 2019. Autopilot online documents. https://nl.mathworks.com/matlabcentral/fileexchange/41490-autopilot-demo-for-arp4754a-do-178c-and-do-331?focused=6796756&tab=model. Accessed: 2019-08-07.

[3] 2019. DHC-2model. http://www.dutchroll.com. Accessed: 2019-10-24.

[4] 2020. Additional Material. https://doi.org/10.5281/zenodo.3872902

[5] 2020. fitctree. https://nl.mathworks.com/help/stats/fitctree.html

[6] 2020. QVtrace. https://qracorp.com/qvtrace/

[7] 2020. Simulink. https://nl.mathworks.com/products/simulink.html

[8] United States. Federal Aviation Administration. 2009. *Advanced Avionics Handbook*. Aviation Supplies & Academics, Incorporated. https://books.google.lu/books?id=2xGuPwAACAAJ

[9] Urtzi Markiegi Ainhoa Arruabarrena Leire Etxeberria Goiuria Sagardui AitorArrieta, Shuai Wang. 2019. Pareto efficient multi-objective black-box test case selection for simulation-based testing. (2019).

[10] Rajeev Alur, Salar Moarref, and Ufuk Topcu. 2013. Counter-strategy guided refinement of GR(1) temporal logic specifications. In *Formal Methods in Computer-Aided Design, FMCAD*. IEEE, 26–33.

[11] Yashwanth Annpureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. 2011. S-TaLiRo: A Tool for Temporal Logic Falsification for Hybrid Systems. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 254–257.

[12] Andrea Arcuri and Lionel C. Briand. 2011. Adaptive random testing: an illusion of effectiveness?. In *International Symposium on Software Testing and Analysis, ISSTA*. ACM, 265–275.

[13] Andrea Arcuri, Muhammad Zohaib Z. Iqbal, and Lionel C. Briand. 2012. Random Testing: Theoretical Results and Practical Implications. *IEEE Trans. Software Eng.* 38, 2 (2012), 258–277. https://doi.org/10.1109/TSE.2011.121

[14] Wolfgang Banzhaf, Peter Nordin, Robert E Keller, and Frank D Francone. 1998. *Genetic programming*. Springer.

[15] Howard Barringer and Dimitra Giannakopoulou. 2003. Proof Rules for Automated Compositional Verification through Learning. In *In Proc. SAVCBS Workshop*. 14–21.

[16] Matthias Bernaerts, Bentley Oakes, Ken Vanherpen, Bjorn Aelvoet, Hans Vangheluwe, and Joachim Denil. 2019. Validating Industrial Requirements with a Contract-Based Approach. In *International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 18–27.

[17] Marcello M. Bersani, Matteo Rossi, and Pierluigi San Pietro. 2016. A tool for deciding the satisfiability of continuous-time metric temporal logic. *Acta Inf.* 53, 2 (2016), 171–206. https://doi.org/10.1007/s00236-015-0229-y

[18] R. V. Borges, A. d'Avila Garcez, L. C. Lamb, and B. Nuseibeh. 2011. Learning to adapt requirements specifications of evolving systems: (NIER track). In *International Conference on Software Engineering (ICSE)*. IEEE.

[19] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. 1984. *Classification and Regression Trees*. Wadsworth.

[20] Christos G. Cassandras and Stéphane Lafortune. 2008. *Introduction to Discrete Event Systems, Second Edition*. Springer. https://doi.org/10.1007/978-0-387-68612-7

[21] Devendra K. Chaturvedi. 2009. *Modeling and Simulation of Systems Using MATLAB and Simulink* (1st ed.). CRC Press, Inc., Boca Raton, FL, USA.

[22] Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. 2010. Adaptive Random Testing: The ART of test case diversity. *J. Syst. Softw.* 83, 1 (2010), 60–66.

[23] Tsong Yueh Chen, Hing Leung, and I. K. Mak. 2004. Adaptive Random Testing. In *Advances in Computer Science - ASIAN*. Springer, 320–329.

[24] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. 2003. Learning Assumptions for Compositional Verification. In *Tools and Algorithms for the Construction and Analysis of Systems TACAS*. Springer, 331–346.

[25] Danil Prokhorov Hisahiro Ito Cumhur Erkan Tuncali, Georgios Fainekos and James Kapinski. 2019. Requirements-driven Test Generation for Autonomous Vehicles with Machine Learning Components. (2019), 137–154.

[26] Luca de Alfaro and Thomas A. Henzinger. 2001. Interface Automata. *SIGSOFT Softw. Eng. Notes* 26, 5 (Sept. 2001), 12. https://doi.org/10.1145/503271.503226

[27] Luca de Alfaro and Thomas A. Henzinger. 2001. Interface Theories for Component-Based Design. In *Embedded Software*, Thomas A. Henzinger and Christoph M. Kirsch (Eds.). Springer, 148–165.

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

[29] P. Derler, E. A. Lee, and A. Sangiovanni Vincentelli. 2012. Modeling Cyber-Physical Systems. *Proc. IEEE* 100, 1 (2012), 13–28. https://doi.org/10.1109/JPROC.2011.2160929

[30] Patricia Derler, Edward A Lee, Stavros Tripakis, and Martin Törngren. 2013. Cyber-physical system design contracts. In *International Conference on Cyber-Physical Systems*. ACM, 109–118.

[31] Gidon Ernst, Paolo Arcaini, Alexandre Donze, Georgios Fainekos, Logan Mathesen, Giulia Pedrielli, Shakiba Yaghoubi, Yoriyuki Yamagata, and Zhenya Zhang. 2019. ARCH-COMP 2019 Category Report: Falsification. *EPiC Series in Computing* 61 (2019), 129–140.

[32] Pappas G.J Fainekos, G.E. 2008. A user guide for TaLiRo.

[33] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. 2011. SpaceEx: Scalable Verification of Hybrid Systems. In *Computer Aided Verification (CAV)*. Springer.

[34] Dimitra Giannakopoulou, Corina S Pasareanu, and Howard Barringer. 2002. Assumption generation for software component verification. In *International Conference on Automated Software Engineering*. IEEE, 3–12.

[35] Dimitra Giannakopoulou, Corina S. Pasareanu, and Colin Blundell. 2008. Assume-guarantee testing for software components. *IET Software* 2, 6 (2008), 547–562. https://doi.org/10.1049/iet-sen:20080012

[36] Dimitra Giannakopoulou, Corina S Pasareanu, and Jamieson M Cobleigh. 2004. Assume-guarantee verification of source code with design-level assumptions. In *International Conference on Software Engineering*. IEEE, 211–220.

[37] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.* 45, 1 (2012), 11:1–11:61. https://doi.org/10.1145/2379776.2379787

[38] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. 1995. What's Decidable about Hybrid Automata?. In *Annual ACM Symposium on Theory of Computing (STOC)*. ACM.

[39] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. 1998. You assume, we guarantee: Methodology and case studies. In *Computer Aided Verification*. Springer, 440–451.

[40] Xiaoqing Jin, Alexandre Donzé, Jyotirmoy V. Deshmukh, and Sanjit A. Seshia. 2013. Mining requirements from closed-loop control models. In *international conference on Hybrid systems: computation and control, HSCC*. ACM.

[41] Xiaoqing Jin, Alexandre Donzé, Jyotirmoy V. Deshmukh, and Sanjit A. Seshia. 2015. Mining Requirements From Closed-Loop Control Models. *IEEE Trans. on CAD of Integrated Circuits and Systems* 34, 11 (2015), 1704–1717. https://doi.org/10.1109/TCAD.2015.2421907

[42] J. Kapinski, J. V. Deshmukh, X. Jin, H. Ito, and K. Butts. 2016. Simulation-Based Approaches for Verification of Embedded Control Systems: An Overview of Traditional and Advanced Modeling, Testing, and Verification Techniques. *IEEE Control Systems Magazine* 36, 6 (2016), 45–64.

[43] Wenchao Li, Lili Dworkin, and Sanjit A. Seshia. 2011. Mining assumptions for synthesis. In *International Conference on Formal Methods and Models*. IEEE, 43–50.

[44] Shahar Maoz, Jan Oliver Ringert, and Rafi Shalom. 2019. Symbolic repairs for GR (1) specifications. In *International Conference on Software Engineering (ICSE)*. IEEE, 1016–1026.

[45] Reza Matinnejad, Shiva Nejati, and Lionel C Briand. 2017. Automated testing of hybrid Simulink/Stateflow controllers: industrial case studies. In *Foundations of Software Engineering*. ACM, 938–943.

[46] Anastasia Mavridou, Hamza Bourbouh, Pierre-Loïc Garoche, Dimitra Giannakopoulou, Tom Pressburger, and Johann Schumann. 2020. Bridging the Gap Between Requirements and Simulink Model Analysis. In *Requirements Engineering: Foundation for Software Quality (REFSQ), Companion Proceedings*. Springer.

[47] Claudio Menghi, Marcello M. Bersani, Matteo Rossi, and Pierluigi San Pietro. 2020. Model Checking MITL formulae on Timed Automata: a Logic-Based Approach. *Transactions on Computational Logic* (2020).

[48] Claudio Menghi, Shiva Nejati, Lionel C. Briand, and Yago Isasi Parache. 2020. Approximation-Refinement Testing of Compute-Intensive Cyber-Physical Models: An Approach Based on System Identification. In *International Conference on Software Engineering (ICSE)*. ACM.

[49] Claudio Menghi, Shiva Nejati, Khouloud Gaaloul, and Lionel C. Briand. 2019. Generating automated and online test oracles for Simulink models with continuous and uncertain behaviors. In *Foundations of Software Engineering (ESEC/SIGSOFT FSE)*. ACM.

[50] Claudio Menghi, Paola Spoletini, Marsha Chechik, and Carlo Ghezzi. 2018. Supporting Verification-Driven Incremental Distributed Design of Components. In *Fundamental Approaches to Software Engineering FASE*. Springer, 169–188.

[51] Claudio Menghi, Paola Spoletini, Marsha Chechik, and Carlo Ghezzi. 2019. A verification-driven framework for iterative design of controllers. *Formal Asp. Comput.* 31, 5 (2019), 459–502. https://doi.org/10.1007/s00165-019-00484-1

[52] Christoph Molnar. 2019. *Interpretable Machine Learning*. https://christophm.github.io/interpretable-ml-book/.

[53] Christoph Molnar. 2019. *Interpretable machine learning*. Lulu. com.

[54] Shiva Nejati, Khouloud Gaaloul, Claudio Menghi, Lionel C. Briand, Stephen Foster, and David Wolfe. 2019. Evaluating Model Testing and Model Checking for Finding Requirements Violations in Simulink Models. In *Foundations of Software Engineering (ESEC/FSE)*. ACM.

[55] Pierluigi Nuzzo, John B. Finn, Antonio Iannopollo, and Alberto L. Sangiovanni-Vincentelli. 2014. Contract-based design of control protocols for safety-critical cyber-physical systems. In *Design, Automation & Test in Europe Conference & Exhibition, (DATE)*. European Design and Automation Association.

[56] Pierluigi Nuzzo, Huan Xu, Necmiye Ozay, John B. Finn, Alberto L. Sangiovanni-Vincentelli, Richard M. Murray, Alexandre Donzé, and Sanjit A. Seshia. 2014. A Contract-Based Methodology for Aircraft Electric Power System Design. *IEEE Access* 2 (2014), 1–25. https://doi.org/10.1109/ACCESS.2013.2295764

[57] Peter J Ramadge and W Murray Wonham. 1987. Supervisory control of a class of discrete event processes. *SIAM journal on control and optimization* 25, 1 (1987), 206–230.

[58] Peter JG Ramadge and W Murray Wonham. 1989. The control of discrete event systems. *Proc. IEEE* 77, 1 (1989), 81–98.

[59] Yinyin Xu Haibo Chen Dave Towey Xin Xia Rubing Huang, Weifeng Sun. 2019. A Survey on Adaptive Random Testing. IEEE Transactions on Software Engineering.

[60] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM.

[61] X. Sun, P. Nuzzo, C. Wu, and A. Sangiovanni-Vincentelli. 2009. Contract-based system-level composition of analog circuits. In *Design Automation Conference*. ACM.

[62] Ashish Tiwari. 2012. HybridSAL Relational Abstracter. In *Computer Aided Verification (CAV)*. Springer.

[63] Ian H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. 2016. *Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques* (4th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.