



PhD-FSTM-2021-060
The Faculty of Sciences, Technology and Medicine

DISSERTATION

Defence held on 15/09/2021 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

by

Khouloud GAALOUL

Born on 30 January 1993 in Nabeul, Tunisia

VERIFICATION OF DESIGN MODELS OF CYBER-
PHYSICAL SYSTEMS SPECIFIED IN SIMULINK

Dissertation defence committee

Dr Shiva NEJATI, dissertation supervisor

Professor, Université du Luxembourg

Dr Fabrizio PASTORE, Chairman

Professor, Université du Luxembourg

Dr Claudio MENGHI, Vice Chairman

Université du Luxembourg

Dr Gregory GAY, member

Professor, Chalmers and University of Gothenburg, Sweden

Dr Arie GURFINKEL, member

Professor, University of Waterloo, Canada

I would like to acknowledge and give my warmest thanks to my supervisor Dr. Shiva Nejati, for her significant effort to make my research work successful. Her dedication, insightful advice and scientific approach kept me constantly engaged with my research.

I would like to express my special gratitude to Prof. Lionel C. Briand, for his support and guidance to accomplish this thesis. I am honored to have had the opportunity to learn from his knowledge and academic excellence.

I owe a deep gratitude to my co-supervisor, Dr. Claudio Menghi for significantly enriching this thesis. His dynamism and unwavering generosity helped me accomplish my tasks.

Special thanks goes to my family and friends for their continuous support throughout my PhD.

Abstract

Recent advances in cyber-physical systems (CPS) have allowed highly available and approachable technologies with interconnected systems between the physical assets and the computational software components [BG11a]. This has resulted in more complex systems with wider capabilities. For example, they can be applied in various domains such as safe transport, efficient medical devices, integrated systems, critical infrastructure control and more. The development of such critical systems requires advanced new models, algorithms, methods and tools to verify and validate the software components and the entire system. The verification of cyber-physical systems has become challenging: (1) The complex and dynamical behaviour of systems requires resilient automated monitors and test oracles that can cope with time-varying variables of CPS. (2) Given the wide range of existing verification and testing techniques from formal to empirical methods, there is no clear guidance as to how different techniques fare in the context of CPS. (3) Due to serious issues when applying exhaustive verification to complex systems, a common practice is needed to verify system components separately. This requires adding implicit assumptions about the operational environment of system components to ensure correct verification. However, identifying environment assumptions for cyber-physical systems with complex, mathematical behaviors is not trivial. In this dissertation, we focus on addressing these challenges.

In this dissertation, we propose a set of effective approaches to verify design models of CPS. The work presented in this dissertation is motivated by ESAIL maritime micro-satellite system, developed by LuxSpace [Lux19], a leading provider of space systems, applications and services in Luxembourg. In addition to ESAIL, we use a benchmark of eleven public-domain Simulink models provided by Lockheed Martin [loc20], which are representative of different categories of CPS models in the aerospace and defence sector. To address the aforementioned challenges, we propose (1) an automated approach to translate CPS requirements specified in a logic-based language into test oracles specified in Simulink. The generated oracles are able to deal with CPS complex behaviours and interactions with the system environment; (2) An empirical study to evaluate the fault-finding capabilities of model testing and model checking techniques for Simulink models. We also provide a categorization of model types and a set of common logical patterns for CPS requirements; (3) An automated approach to synthesize environment assumptions for a component under analysis by combining search-based testing, machine learning and model checking procedures. We also propose a novel technique to guide the test generation based on the feedback received from the machine learning process; and (4) An extension of (3) to learn more complex assumptions with arithmetic expressions over multiple signals and numerical variables.

Contents

List of Figures	5
List of Tables	7
1 Introduction	1
1.1 Context	1
1.2 Challenges	2
1.3 Research Contributions	3
1.4 Dissertation Outline	4
2 Background	5
2.1 System Modeling for CPS	5
2.2 MATLAB/Simulink	6
2.3 Model-Based Verification of CPS	8
2.4 Test Oracles	12
2.5 Supervised Machine Learning	14
3 Generating Test Oracles for Simulink Models with Continuous and Uncertain Behaviours	19
3.1 Motivation and Challenges	20
3.2 Approach Overview	22
3.3 Evaluation	30
3.4 Related Works	34
3.5 Conclusions	34
4 Evaluating Model Testing and Model Checking for Finding Requirements Violations in Simulink Models	37
4.1 Simulink Benchmark	38
4.2 Our Model Checking Technique	43
4.3 Our Model Testing Technique	44
4.4 Empirical Evaluation	45
4.5 Lessons Learned	49
4.6 Conclusions	50

5 Mining Assumptions for Software Components using Machine Learning	51
5.1 Motivation and Challenges	52
5.2 Approach Overview	53
5.3 Assumption Generation Problem	56
5.4 Implementation	58
5.5 Evaluation	63
5.6 Discussion and Threats To Validity	66
5.7 Conclusion	67
6 Combining Genetic Programming and Model Checking to Generate Environ-	
ment Assumptions	69
6.1 Motivation and Challenges	71
6.2 Approach Overview	74
6.3 Assumption Generation	79
6.4 Evaluation	84
6.5 Related Works	97
6.6 Conclusion	99
7 Conclusions & Future Work	101
7.1 Summary	101
7.2 Future Work	102
Bibliography	105

List of Figures

1.1	Autopilot model	2
2.1	Diagrammatic layout of CPS	6
2.2	Schematic view of the system verification	6
2.3	Example of Simulink model: (a) Autopilot model, (b) Plant subsystem: DeHavilland Beaver Airframe, (c) Controller subsystem: Roll controller	7
2.4	An example of Input/Output signals of the Autopilot Simulink model shown in Figure 2.3a	7
2.5	Schematic view of Model Checking	10
2.6	Test oracle in software testing process	12
2.7	Supervised machine learning model: The algorithm learns a model from labeled training data. The model is used to make predictions on the output y_n of the inputs x_n .	14
2.8	An example of decision tree for distinguishing papayas	15
2.9	Genetic Programming Flowchart	16
3.1	Three simulation outputs of our ESAIL case study model indicating the error signal $\ \vec{\mathbf{q}}_{\text{real}}(\mathbf{t}) - \vec{\mathbf{q}}_{\text{target}}(\mathbf{t})\ $. The signal in (a) passes R4 in Table 3.1, but those in (b) and (c) violate R4 with low and high severity, respectively.	20
3.2	Overview of SOCRaTes, our automated oracle generation approach.	22
3.3	Signals $\ \vec{w}_{\text{sat}}\ $ for the w_{sat} output of ESAIL. The solid-line signal is generated by ESAIL with no uncertainty, and the dashed-line signal is generated when the S2N ratios in Table 3.3 are applied to the ESAIL inputs.	23
3.4	A set of signals $\ \vec{w}_{\text{sat}}\ $ for the output w_{sat} of ESAIL with uncertain parameters (i.e., when the sun sensor and magnetometer parameters are specified as in Table 3.3).	24
3.5	Plots reporting (a) the size of the RFOL formulas, (b) the number of blocks and connections of the oracle models and (c) the time took SOCRaTEs to generate the oracles.	32
3.6	Test execution time on models without oracles (M), models with oracles (M + M_φ) with <i>threshold</i> = 0 and models with oracles (M + M_φ) with <i>threshold</i> = -1 for (a) Autopilot, (b) ESAIL without uncertainty and (c) ESAIL with uncertainty.	32
4.1	Simulink Model Verification: (a) Model testing and (b) Model checking.	38
4.2	Generic structure of (a) open-loop and (b) feedback-loop CPS models	40
5.1	An overview on the EPICuRus framework.	54

5.2	Example of classification tree constraining some of the inputs of the Autopilot running example.	56
5.3	Comparison of the Test Case Generation Policies.	66
6.1	Example of Input/Output signals of the AC model.	71
6.2	EPIcuRus framework overview.	76
6.3	Syntactic rules of the grammar that defines the assumptions on control points. The symbol separates alternatives, const is a constant value, and cp is a variable that refers to a control point.	81
6.4	Syntax tree associated with an individual.	83
6.5	Comparing GP, DT, and RS. The box plots show the coverage value of GP, DT, and RS (labels on the bottom of the figure). Diamonds depict the average. The value below the box plot is the percentage of runs, across all the runs, in which the technique was able to compute a sound assumption.	88
6.6	Comparing GP, DT, and RS. The box plots show the coverage value of GP, DT, and RS for the input profiles IP, IP' and, IP'' (labels on the bottom of the figure). Diamonds depict the average. The value below the box plot is the percentage of runs, across all the runs of each input profile, in which the technique was able to compute a sound assumption.	90

List of Tables

3.1	Requirements for the satellite control system (ESAIL) developed by LuxSpace.	21
3.2	Signals variables of the ESAIL model.	21
3.3	Uncertainty in Autopilot: The values of the magnetometer type and the sun sensor accuracy parameters are given as ranges (middle column). The noise values for the magnetometer and sun sensor inputs are given in the right column.	24
3.4	Translating the SFFO formulae into Simulink Oracles.	29
3.5	Important characteristics of our case study systems (from left to right): (1) name, (2) description, (3) number of blocks of the Simulink model of each case study (#Blocks), (4) number of requirements in each case study (#Reqs) and (5) total number of blocks necessary to encode the requirements (#BIReqs)	31
4.1	Important characteristics of our benchmark Simulink models (from left to right): (1) model name, (2) model description, (3) model type, and (4) number of atomic blocks in the model.	39
4.2	Example requirements for the TwoTanks and Autopilot models.	40
4.3	Translation of Signal Temporal Logic [MN04] into quantitative fitness functions to be used in the model testing approach in Figure 4.1(a)	42
4.4	Temporal patterns in STL translations of our benchmark requirements.	43
4.5	Temporal patterns used in the requirements formalisations of each Simulink benchmark model.	44
4.6	Comparing the abilities of model testing and model checking in finding requirements violations for Simulink models.	47
4.7	Comparing the time performance of model testing and model checking.	48
5.1	Generating the predicates of the constraint.	61
5.2	Identifier, name, description, number of blocks of the Simulink model (#Blocks), number of inputs (#Inputs) and number of requirements (#Reqs) of our study subjects.	63
6.1	Name of the Input, number of input signals in the virtual vector (NS), and description of the input.	72
6.2	Example requirements for the attitude control component.	72
6.3	Parameters of EPICuRus (EP) and its Genetic Programming algorithm (GP).	80

6.4	Identifier (ID), name, description, number of blocks (#Bk), inputs (#In), outputs (#Out), simulation time (ST), number of requirements (#Reqs), and the number of requirements we used to answer research question RQ1, i.e., they pass the sanity check (#Reqs within round brackets) of each Simulink model of the components of our study subjects.	86
6.5	Values for the parameters of Table 6.3 used for RQ1.	87
6.6	Number of runs, among the 100 runs of each requirement-profile combination, in which GP, DT and RS were able to compute a sound assumption	89
6.7	Values for the parameters of Table 6.3 used to assess whether EPICuRus can learn assumptions similar to the one manually defined by engineers.	91
6.8	Minimum and maximum value of each term of exp.	91
6.9	The value C of the coefficient of the term T_i in exp, The number N of runs in which EPICuRus was able to learn T_i in P_1 and P_2 . The percentage S of runs in which the sign of T_i was correct. The average D and the maximum MaxD of the difference between the coefficient of T_i of A_1 and the one returned by EPICuRus.	92
6.10	Values for the parameters of Table 6.3 used for RQ2-2.	96

Chapter 1

Introduction

1.1 Context

The development of cyber-physical systems (CPS) [BG11b, SWYS11, Lee08] relies on early function modeling of the system and its environment. These models typically capture dynamical systems. For example, they may be mathematical models capturing movements of a physical object. They may also specify a software controller that interacts with a physical process to respectively control the movements of the object or the progression of the process over time. A key and common feature of these models is that they typically consist of time-varying and real-valued variables and functions.

To capture CPS dynamical systems, CPS industry typically uses development and simulation languages among which Simulink is widely known. Simulink is used by more than 60% of engineers for simulation of CPS [ZJKK17, BDN17], and is the prevalent modeling language in the automotive domain [MNBB16a, ZSM12]. Simulink appeals to engineers since it is particularly suitable for specifying mathematical models and dynamic systems, and further, it is arguably an effective way of design time testing for engineers.

To avoid ripple effects from defects and to ensure that failures are identified as early as possible, it is paramount for the CPS industry to ensure that system models satisfy their functional safety requirements. As mandated by safety certification standards [C⁺00], these requirements must be specified, and be used as the main authoritative reference to demonstrate system behavior correctness. The ultimate goal of software verification techniques is to provably demonstrate the correctness of the Simulink model against these requirements.

In this dissertation, we address the problem of verification of behavioral design models of CPS specified in Simulink. This dissertation presents a set of approaches using search-based testing, model checking and machine learning to automate the verification and testing of CPS. The work presented in this dissertation has been done in collaboration with LuxSpace [Lux19], a leading provider of space systems, applications and services in Luxembourg, and QRA Corp [qra19], a verification tool vendor to the aerospace, automotive and defence sectors in Canada.

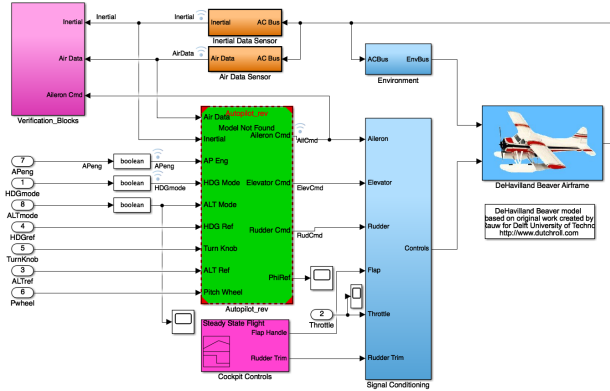


Figure 1.1: Autopilot model

We present and describe the case study systems used to evaluate our approaches. Specifically, we use the ESAIL maritime micro-satellite, a system developed by LuxSpace [Lux19], our industrial partner, in collaboration with ESA [ESA20b] and ExactEarth [exa20]. ESAIL aims at enhancing the next generation of space-based services for the maritime sector. During the design phase of the satellite (i.e., development phases B-C [ESA20a]), the control logic of the ESAIL software is specified as a Simulink [mat19] model. We also used a set of public-domain benchmark of eleven industrial Simulink models provided by Lockheed Martin [loc20], which are representative of different large-scale CPS models in the aerospace and defense sector. Our benchmark has an autopilot system that comes with a full six degree of freedom simulation of an airplane, a neural network model with two hidden layers, a PID controller, a finite state machine, etc. These systems include components with various computational categories which are responsible for the functional and behavioural variations in CPS models. Figure 1.1 shows Autopilot, an example of our industrial Simulink models.

1.2 Challenges

Our primary goal in this thesis is to develop automated verification tools for CPS. To achieve this goal, we address the following challenges:

- Existing approaches to testing of Simulink models have largely focused on automated generation of test suites [ND15, HMSY13, OKN14], while test automation requires also *automated oracles* [BHM⁺15] to assess whether a test has passed or failed. The dynamical behaviour, the time-varying features and the interactions of the system with the environment remain the main challenges for *automated oracles* in the context of CPS.
- Different approaches to verification and testing of Simulink models have been proposed in the literature [Ham08, BBB⁺12b, MNBB18, AFS⁺13]. There are two mainstream approaches to verify CPS simulink models: *Model testing*, that attempts to identify failures in models by executing them for a number of sampled test inputs; and *Model checking* that attempts to exhaustively check the correctness of models against some given formal properties. The strengths and weaknesses of both approaches depend highly on the category of the model type and the recurring logical pattern in the model requirement. However, there are no systematic comparisons between these approaches, and hence, it remains unclear how *Model checking* or *Model testing* may differ or complement one another in the context of CPS.

- Irrespective of the techniques applied by commercial and industry-strength tools for testing and verification of Simulink models, exhaustive verification is generally undecidable for cyber-physical or hybrid systems [HKPV95]. These systems are generally far too complex to be verified exhaustively and in their entirety [KDJ⁺16, NGM⁺19]. As a result, in practice, exhaustive verification techniques (e.g., [FGD⁺11, Tiw12]) can only be applied to some selected individual components or algorithms that are amenable to exhaustive verification and are used within a larger system. This practice may result in spurious failures due to the violation of environment assumptions. Environment assumptions are the expectations that a system or a component makes about its operational environment and are often specified in terms of conditions over the inputs of that system or component. However, environment assumptions are rarely fully documented for software systems [SMP⁺18] and the manual identification is tedious and time-consuming. This problem is exacerbated for cyber-physical systems (CPS) that often have complex, mathematical behavior.

1.3 Research Contributions

In this dissertation, we address the challenges for the verification of design models of CPS. Specifically, we propose the following contributions:

1. Socrates: An automated approach to translate CPS requirements specified in a logic-based language into test oracles specified in Simulink. Our approach generates oracles that: (i) check test outputs in an online manner to stop expensive test executions as soon as a failure is detected; (ii) handle time- and magnitude-continuous CPS behaviors; (iii) provide a quantitative degree of satisfaction or failure measure instead of binary pass/fail outputs; and (iv) are able to handle uncertainties due to CPS interactions with the environment. This contribution has been published as a conference paper [MNGB19] and is presented in Chapter 3
2. We report on an empirical study to evaluate capabilities of model testing and model checking techniques in finding faults in Simulink models: (i) We provide a categorization of CPS model types and a set of common logical patterns in CPS functional requirements using an industrial benchmark consisting of Simulink models from the CPS industry. We further formalize the textual requirements in a logic-based requirements language and identify some common patterns among the CPS requirements in the benchmark. (ii) We present the results of applying our model testing and model checking techniques to the Simulink benchmark. We evaluate the fault finding abilities of both techniques. (iii) We provide some lessons learned by outlining the strengths and weaknesses of model testing and model checking in identifying faults in Simulink models. As these two approaches provide complementary benefits, we believe that integrating them in a comprehensive verification framework can result in an effective testing method. We further propose some guidelines as to how the two approaches can be best applied together. This contribution has been published as a conference paper [NGM⁺19] and is presented in Chapter 4
3. Epicurus: An automated approach to 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. This contribution has been published as a conference paper [GMN⁺20] and is presented in Chapter 5.

4. The technique developed in Chapter 5 in collaboration with QRACorp [qra19], and assessed on the models provided by Lockheed Martin, cannot learn the complex assumptions for the AC component of ESAIL, an industrial and complex model of a CPS provided by LuxSpace. This was a significant limitation of the previous work that required substantial improvement. Therefore, we improve Epicurus in several ways. First, we rely on GP, rather than decision trees to (a) learn assumptions for complex CPS models involving signal and numerical variables; and (b) learn assumptions that include arithmetic expressions defined over multiple variables. We adapt, modify, and customize GP to learn such complex assumptions. Second, we define a new methodology to translate assumptions on control points (learned using GP), into assumptions over signal values. Third, we evaluate the new version of EPICuRus to show that it substantially improves the previous version. We also evaluate the usefulness of EPICuRus on the AC component of ESAIL. Finally, we identify a trade-off between soundness and coverage of environment assumptions and demonstrate the flexibility of our approach in prioritizing either of these criteria. This contribution is presented in Chapter 6.

1.4 Dissertation Outline

Chapter 2 provides some fundamental background on software verification approaches including *Model testing* and *Model checking* and assumption inference approaches.

Chapter 3 presents our approach to generate automated and online oracles for Simulink models with continuous and uncertain behaviours.

Chapter 4 describes how we empirically evaluate *Model testing* and *Model checking* for finding requirements violations in Simulink models.

Chapter 5 introduces our automated approach to generate environment assumptions for software components using machine learning decision trees.

Chapter 6 presents our extended approach to synthesize assumptions by combining *Model testing* and *Genetic programming*.

Chapter 7 summarizes the thesis contributions and discusses perspectives on future works.

Chapter 2

Background

This chapter presents the background concepts that we consider throughout this dissertation. This chapter is organized as follows: Section 2.1 defines Cyber-Physical Systems (CPS) and Model-Based Design (MBD), and further describes the challenges of system modeling for CPS. Section 2.2 defines CPS simulink models. Section 2.3 describes the existing techniques applied in the context of software verification (e.g., model testing and its search algorithms, model checking). Section 2.4 defines the First-Order Logic of Signals (SFO) language and introduces the oracle generation technique that we consider in this dissertation. Section 2.5 introduces few main concepts of supervised machine learning including Decision Trees (DT) and Genetic Programming (GP).

2.1 System Modeling for CPS

Cyber-physical systems (CPS) [BG11b, SWYS11, Lee08] are characterised by integrating computation, networking, and physical processes into one whole system of components that deeply interact among each others to provide certain features. Figure 2.1 shows a diagrammatic layout of cyber-physical systems. CPS applications arguably have a positive impact on several domains. For example, they can be applied to critical medical devices such as monitoring equipment. They can be applied to advanced autonomous systems where transportation systems could benefit considerably from the embedded intelligence that guarantees the safety and efficiency of these systems. While cyber-physical systems have a huge impact on multiple areas of applications, they must be robust enough and must be adapted to overcome system failures. Therefore, these critical systems require advanced methods and tools to verify and validate the software components and the entire system.

Model-based design (MBD) [CGP01, FR07, JCL11] is a powerful design technique for CPS which establishes effective communication throughout the design process. The design techniques include creating mathematical abstractions and modeling [Wai09, Cha10, Mar97] of the physical systems, usually specified in the form of systems of differential equations or Laplace transfer functions. These functions describe the system properties and interactions within the system. They may also include formal models of computation, software synthesis, verification, validation and testing

Given the sophisticated characteristics of cyber-physical systems, model-based design goes through several iterations of verification and refinement rounds to gain enough confidence in the model behaviors and prepare it for code generation. Figure 2.2 depicts a schematic view of model-based

verification, one of the main steps of model-based design. During this phase, engineers iteratively perform several rounds of modeling and verification until the requirements are satisfied [ZC06]. These requirements describe what needs the system has to meet and what it shall not do. During model-based verification, the model of the system is created (or refined) and the requirements are formalized based on the CPS specification. Then, the verification step checks the system model against the requirements, and finally returns *No violation found* when the model satisfies these requirements. Otherwise, it returns *Violation found*.

In the CPS domain, models are characterised by dynamical behaviors and exhibit time-varying changes depending on their modeling paradigm. They can be time-continuous or time-discrete and they can be magnitude-continuous or magnitude-discrete. To be able to capture the dynamical behaviour of the physical plants and the interactions with the software systems, models of complex CPS often build on time-continuous and magnitude-continuous paradigms. Contrary to the models that are designed to capture only software computations, these models are often described using time-discrete magnitude-discrete abstractions.



Figure 2.1: Diagrammatic layout of CPS

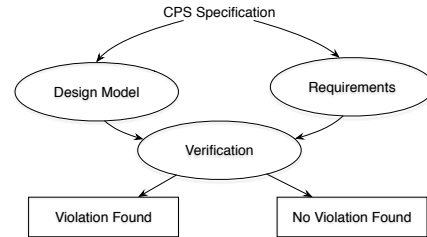


Figure 2.2: Schematic view of the system verification

2.2 MATLAB/Simulink

MATLAB/Simulink is an advanced platform for modeling, simulation and code-generation of dynamical systems [sim21]. Simulink is a toolbox of MATLAB which provides an executable data-flow driven block diagram language. MATLAB and Simulink enable the design and development of a wide range of advanced products in the automotive, aerospace, telecommunications and many other areas. A high number of engineers worldwide analyze and design the systems and products of their needs for machine learning, signal processing, image processing, computer vision, communications, control design, robotics, and much more using Simulink. Figure 2.3 shows Autopilot model as an example of Simulink model.

Simulink provides a library of blocks, input/output ports and connections. Blocks typically represent operations and constants of a system. Ports are means to specify any signal data that traverse the block. Connections establish data-flows between ports. The system's behaviour can be described in terms of input sequences, actions, conditions and flows of data from input to output by building a Simulink model using the library set. Simulink models are structured into a hierarchy of subsystems, each one accepting a certain number of input signals and producing a certain number of output signals. For example, the model of Autopilot shown in Figure 2.3a has 587 blocks, 623 connections and 514 ports. Blocks, connections, and ports are organized in five subsystems modeling respectively: the behavior of the aircraft, the behavior of its environment (e.g., the atmosphere), Autopilot, the cockpit controls, how the outputs of Autopilot and the cockpit controls affect the inputs of the aircraft, and two subsystems modeling the sensors of the aircraft. The subsystem in Figure 2.3b has two input and one output signals. Each subsystem can also include other subsystems.

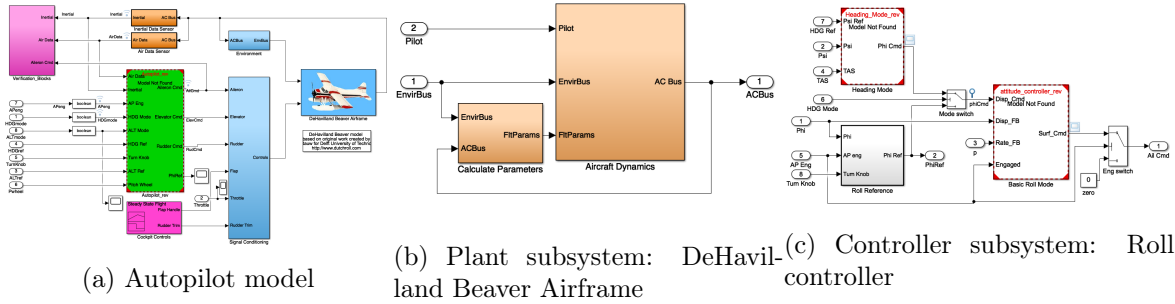


Figure 2.3: Example of Simulink model: (a)Autopilot model, (b) Plant subsystem: DeHavilland Beaver Airframe, (c) Controller subsystem: Roll controller

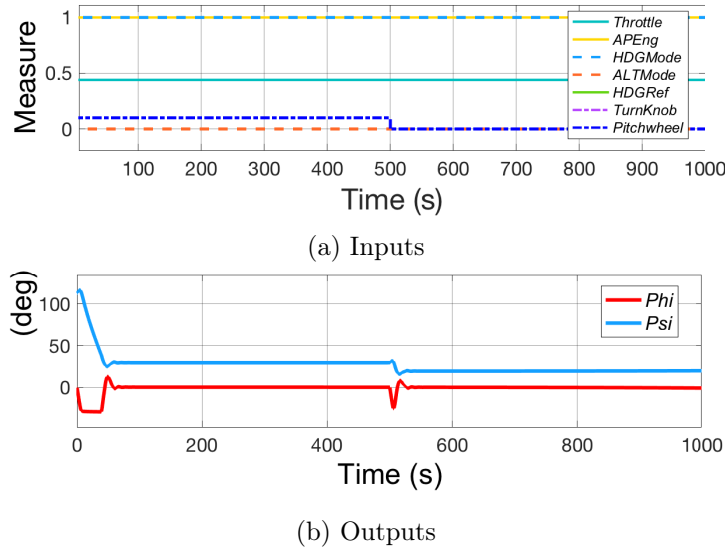


Figure 2.4: An example of Input/Output signals of the Autopilot Simulink model shown in Figure 2.3a

For example, the control subsystem (colored in green) of Autopilot model shown in Figure 2.3a contains several subsystems, including Roll controller subsystem shown in Figure 2.3c. Simulink blocks inside these subsystems perform a sequence of computations on the subsystem's input values and returns the computation values as output. These computations eventually contribute to the values of the output(s) of the model.

To simulate a Simulink model M , the simulation engine receives signal inputs defined over a time domain and computes signal outputs at successive time steps over the same time domain used for the inputs. A *time domain* is a non-singular bounded interval $\mathbb{T} = [0, b]$ 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 set $\bar{u} = \{u_1, u_2 \dots u_m\}$ of input signals and produces a set $\bar{y} = \{y_1, y_2 \dots y_n\}$ of output signals such that each $o_i \in \bar{y}$ corresponds to one model output.

For example, Fig. 2.4a shows a test input for Autopilot example, where signals are defined over the time domain $[0, 10^3]$, and Fig. 2.4b shows the corresponding test output. Specifically, Fig. 2.4a shows input signals for Autopilot. Each of these inputs is related to a command received from the pilot through the instruments in the cockpit (i.e., $APEng$, $HDGMode$, $ALTMode$ represent the status of Autopilot engaging mode enabler; $HDGRef$ represents the desired value of the heading

angle; *Throttle* represents the power applied to the engine; *PitchWheel* represents the degree of adjustments applied to the aircraft pitch; and *TurnKnob* represents the desired value of the roll angle). Fig. 2.4b shows the output signals of Autopilot, namely *Psi* and *Phi* that respectively represent the heading and the roll angles [Adm09]. The Simulink simulation engine takes around 30s to simulate the behavior of the system over the time domain $[0, 10^3]$.¹

Simulink uses numerical algorithms [Sim19] referred to as *solvers* to compute simulations. There are two main types of solvers: *fixed-step* and *variable-step*. *Fixed-step* solvers generate signals over discretized time domains with equal-size time-steps, whereas *variable-step* solvers (e.g., Euler, Runge-Kutta [Atk08]) generate signals over continuous time domains with a time-step size that can vary from a step to another, depending on the model dynamics and the error tolerance.

2.3 Model-Based Verification of CPS

A wide range of promising approaches to model-based verification of Simulink models has been proposed from exhaustive verification to empirical experiments with finite set of model executions. Existing techniques can be broadly categorized into two groups: *model testing techniques* and *model checking techniques*. In the following, we introduce each verification method and we describe their basic pros and cons.

Model Testing

Model testing [BNSB16, MNBB16b, MNBB16c, MNB⁺13, MNB⁺15a, ZC08] is a widely known technique used mainly for testing purposes [BNSB16, MNBB18, AFS⁺13]. This technique relies on model simulation and aims to generate test scenarios and test oracles that violate a given requirement. Model-based testing starts by modeling the system and its environment. Then, the test inputs are generated and executed on the system under test. Finally, a verdict is assigned to the test inputs.

Evolutionary search algorithms [WRDM96, Win09, Win10, Tal09, Yan11] have been applied on Simulink models to generate test inputs that violate the requirement. Specifically, search algorithms sample the input space, selecting the fittest test inputs, i.e., test inputs that (likely) violate or are as close to violate the requirement under analysis. Then, the fittest test inputs is evolved using genetic operators to generate new test inputs and reiterate through the search loop [Luk13]. The test inputs are expected to eventually move towards the fittest regions in the input space (i.e., the regions containing fault-revealing test inputs).

To evaluate how good a test input is, fitness functions are defined a fitness values are computed on the test input. The robustness is one of the widely know metrics [FP09], used to compute fitness values using the model outputs obtained by executing the model under test for sampled test inputs. When the robustness value is positive, the value shows how far a test input is from violating a requirement and when it is negative, its value shows how severe the failure revealed by a test is. For complex discrete-continuous Simulink models, detecting faults in the model is not trivial and requires techniques with high fault-revealing ability [PHPS03].

Model testing applies search operators, that can be either *exploitative* or *explorative*. *Exploitative* search operators evolve elements by making small modifications using a tweak operator while *explorative* search operators make large changes allowing jumps in the search space. An example of *exploitative* single-state search algorithm is Hill-Climbing (HC) [MH93, BHSL17]. As described in the algorithm 1 of HC, we iteratively generate new solutions *NS* by tweaking the current solution *CS*. For example, if the candidate solution *CS* is encoded by a single real value *x*, the Tweak

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

operator shifts CS in the input space by adding value x' to x . The value of x' is typically selected from a normal distribution with mean $\mu = 0$ and variance σ^2 . The new solutions are selected only if their fitness values are better than the current ones. An example of a pure *explorative* algorithm is Random Search (RS) [BB12, Rog72, MH93]. In RS, there is no tweak operation. A new solution is generated randomly at each iteration without any regard to the current solution.

Algorithm 1 Hill Climbing Algorithm.

```

1:  $I$  : Input Space
2:  $CS \leftarrow$  initial candidate solution in  $I$ 
3: repeat
4:    $NS \leftarrow$  Tweak(Copy( $CS$ ))
5:   if  $f(NS) < f(CS)$  then
6:      $CS \leftarrow NS$ 
7: until  $CS$  is the ideal solution or we have run out of time
8: return  $CS$ 

```

There are algorithms in between pure *exploitative* and pure *explorative*, i.e., Simulated Annealing (SA) [VLA87, BT⁺93, Rut89, Dav87]. Similarly to HC, SA replaces the current solution CS with a new solution NS if the current one has a worse fitness value. However, unlike HC, in some cases, SA may replace the current solution with a new solution even if the current solution has a better fitness value (see algorithm 2). This situation occurs only if another condition based on temperature t value holds. Temperature t is initialised at the beginning of search, and decreases over time. This means that SA replaces the current solution with a new solution more often at the beginning, having more explorative behaviour, and turns to have more exploitative behaviour over time.

Algorithm 2 Simulated Annealing Algorithm.

```

1:  $I$  : Input Space
2:  $t \leftarrow$  temperature, initially a high number
3:  $CS \leftarrow$  initial candidate solution in  $I$ 
4: repeat
5:    $NS \leftarrow$  Tweak(Copy( $CS$ ))
6:   if  $f(NS) < f(CS)$  or if a random number chosen from 0 to 1  $< e^{\frac{f(NS)-f(CS)}{t}}$  then
7:      $CS \leftarrow NS$ 
8:   Decrease  $t$ 
9:   if  $f(NS) < f(CS)$  then
10:     $CS \leftarrow NS$ 
11: until  $CS$  is the ideal solution or we have run out of time
12: return  $CS$ 

```

Since model testing works by sampling test inputs from the input search space of the model under test, it requires the value ranges of the model input variables to be provided. The value ranges of model input variables are generally extracted from the model description documents.

Intensive research has proved that model-based testing ensures coverage of basic behaviours of the system under test using a variety of cost-effective test generation strategies and model coverage criteria. The main advantage of applying model-based testing is that the overall test design time is reduced and that a variety of test suites can be generated from the model using different generation methods. Furthermore, Model testing does not suffer from applicability and scalability issues encountered by exhaustive verification methods thanks to its black-box nature. However, it can only

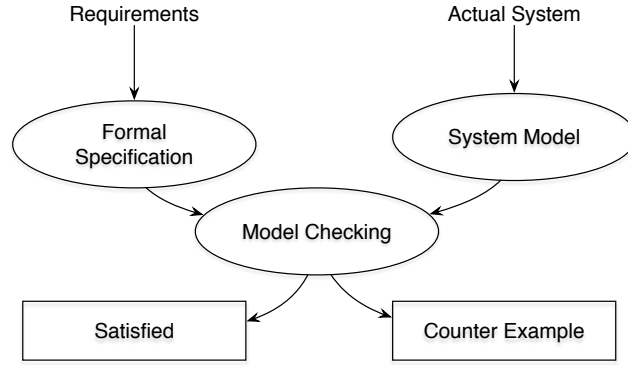


Figure 2.5: Schematic view of Model Checking

show the presence of failures and not their absence. The effectiveness of model testing techniques relies heavily on their underlying heuristics and search guidance strategies. Since there is no theoretically proven way to assess and compare different search heuristics, model testing techniques can only be evaluated empirically.

Model Checking

Formal verification methods aim to establish system correctness mathematically during the design process. These methods generally tend to reduce the verification time dramatically, which makes them one of the highly recommended verification techniques for industrial cyber-physical systems. Model checking (MC) [CJGK⁺18, AKLP10, BBB⁺12a, MBC10] is a formal verification method that attempts to exhaustively verify the correctness of models against some given formal properties. It has a long history of application in software and hardware verification and it has been previously used to detect faults in Simulink models [BBB⁺12a, BBC⁺06]. Figure 2.5 shows an overview of the model checking technique.

Model checking has three main phases. It starts with *modeling* phase where the Simulink model is translated into the language of some model checking tool (e.g., Satisfiability Modulo Theories (SMT) solvers [BBB⁺12a, HDE⁺08, Mil09, SSC⁺04, AIER15]). This task can be a simple compilation or may require a few rounds of abstractions in case of limited memory.

The second phase of model checking is *specification*, where the property is specified in a logical formalism accepted by the model checker. For software systems, a commonly useful logic for specifying properties is temporal logic [RU12]. Temporal logics are widely known for capturing the behaviour of the system that evolves over time. In fact, they are able to implicitly express the ordering of events in time for systems including CPS where the model inputs and outputs are signals (i.e., functions over time). Moreover, the language used to formalize CPS requirements has to be defined over signal variables.

Let $[a, b]$ s.t. $b \geq a \geq 0$ be an interval of real numbers. We denote signals by s and define them as $s : [a, b] \rightarrow \mathcal{R}$ where \mathcal{R} is the signal range that can be boolean, enum or an interval of real numbers. We denote signals with a boolean or enum range by $s^{\mathbb{B}}$ and those with real intervals or numbers by $s^{\mathbb{R}}$. A very common way to formally specify model requirements is Signal Temporal Logic (STL) [MN04], an extension of the well-known Linear Temporal Logic (LTL) [Pnu77] with *real-time* temporal operators and *real-valued* constraints. The syntax of STL is given below.

$$\begin{aligned} \varphi &::= \perp \mid \top \mid s^{\mathbb{B}} \mid \mu \text{ rel-op } 0 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 U_{[a,b]} \varphi_2 \\ \mu &::= \mathbf{n} \mid s^{\mathbb{R}} \mid \mu_1 \text{ math-op } \mu_2 \mid f(\mu) \mid (\mu) \end{aligned}$$

where $s^{\mathbb{B}}$ is a boolean-valued signal, $s^{\mathbb{R}}$ is a real-valued signal, **rel-op** is a relational operator (i.e., \geq , $>$, $<$, \leq , $=$, \neq), **math-op** is a numeric operator (i.e., $+$, $-$, \times , $/$), \mathbf{n} is a positive real number (\mathbb{R}_0^+) and $U_{[a,b]}$ is a real-time temporal operator. In the above grammar, f indicates a mathematical function applied to μ such as logarithm or trigonometry functions.

The semantic of STL is described in the literature [MN04]. Briefly, φ formulas, except for (μ **rel-op** 0), are temporal logic formulas where U is the temporal until operator. In STL, the temporal until operator is augmented with an interval $[a, b]$ of real numbers indicating that the until operator is applied to the signal segment between time instants a and b . Finally, the temporal eventually operator F can be written based on the until operator as follows: $F_{[a,b]}\varphi = \top U_{[a,b]}\varphi$, and the globally operator G can be written as $G_{[a,b]}\varphi = \neg F_{[a,b]}\neg\varphi$. Note that when we write a temporal operator without specifying a time interval, we assume that the operator applies to the time interval of its underlying signals. For example, suppose we have signals defined over time interval $[0, T]$, we then write $G\varphi$ as a shorthand for $G_{[0,T]}\varphi$.

The third phase of model checking is *verification*. During this phase, the model checking tool analyzes the validity of the property on the system model and returns the verdict as a result of the analysis. If the property is violated, a counterexample input trace that violates the property is generated. The returned trace helps the designer track the faults that may be in the system modeling or in the property specification. However, this phase may fail to terminate successfully due to the memory problems which is usually caused by the model complexity. In this case, additional abstractions are required during the *modeling* phase as a common practice.

Despite the large volume of academic research on software testing and verification, there are relatively few commercial and industry-strength tools for the verification of Simulink models. Among the commercial model checking tools for Simulink models (i.e., QVtrace and SLDV), QVtrace [gra19] is a recent commercial tool that builds on the ideas from SMT-based model checking. Specifically, the formal property together with the model are translated into logical constraints that can be fed into SMT solvers. These solvers, then, attempt to verify that given formal properties hold on the models, or otherwise, they generate counter-examples demonstrating the presence of faults in the models. SLDV, similarly to QVtrace, is a SMT-based model checker. QVtrace is a standalone product developed by QRA [gra19], a Canada-based company specialized in the development of enterprise tools for early-stage validation and verification of critical systems. QVtrace is more recent compared to SLDV and it has a wider range of features and benefits from a well-designed and usable interface. In contrast to SLDV that can only be used with Prover [Pro], QVtrace can be combined with various well-known SMT solvers and theorem provers such as Z3 [DMB08a] and Mathematica [mat]. These model checking tools have been successfully used in practice to verify complex Simulink models.

The main challenge of model checking when applied to Simulink models is that these models often capture continuous dynamic and hybrid systems [Alu15]. It is well-known that model checking such systems is in general undecidable [HKPV98, ACH⁺95, Alu11]. The translation of continuous systems into discrete logic often has to be handled on a case-by-case basis and involves loss of precision which may or may not be acceptable depending on the application domain. Further, industrial Simulink models often contain features and constructs that cannot be easily translated into low-level logic-based languages. Such features include third-party code (often encapsulated in Matlab S-Functions) and non-algebraic arithmetics (e.g., trigonometric, exponential, and logarithmic functions). Nevertheless, model checking, when applicable, can both detect faults and demonstrate lack thereof.

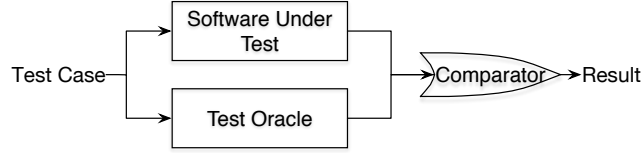


Figure 2.6: Test oracle in software testing process

2.4 Test Oracles

A test oracle [BHM⁺14, NDB13b] is a procedure that determines the correct or incorrect behaviors of the System Under Test. This technique is broadly applied to test the soundness of the program output for some given test cases. It can be in the form of a software system or a specification document. Figure 2.6 shows the schematic view of test oracles in the process of testing. Conceptually, test cases are provided to the test oracle and the program under test. Then, the output of the two components are compared to determine whether the program behaves correctly for the test cases. When the testing result shows a discrepancy between the program and the result, this indicates a defect in the result.

Manual test oracles typically use the program specifications to decide what a correct behaviour of the program should be. Such testing practices, however, may be erroneous. Ideally, automated oracles generated from the specifications of programs or modules guarantee that the output of the oracle conforms with the specification.

In the context of model-based testing, a test oracle is a predicate that determines whether a given test input passes or fails a given requirement of the behaviour of the system under test. We formally define a test oracle. We specifically discuss test oracles for partial Simulink models M_p where simulating the model M_p for a given test input I produces a set of k alternative signals for each output of M_p .

Definition 1 Let M_p be a Simulink model under test, and let I be a test input for M_p defined over the time domain \mathbb{T} . Let φ be a logical formula formalizing a requirement of M_p . Suppose $\{\bar{y}_1, \bar{y}_2 \dots \bar{y}_k\} = H_p(\bar{u}, M_p)$ are the simulation results generated for the time domain \mathbb{T} . We denote the oracle value of φ for test input I over model M_p by $oracle(M_p, I, \varphi)$ and compute it as follows:

$$oracle(M_p, I, \varphi) = \min_{O \in \{\bar{y}_1, \bar{y}_2 \dots \bar{y}_k\}} \llbracket \varphi \rrbracket_O$$

Specifically, $oracle(M_p, I, \varphi)$ indicates the fitness value of the test input I over model M_p and evaluated against requirement φ .

The oracle output is a value in $[-1, 1]$. As defined above, for a partial model, the oracle computes the minimum value of φ over every test output set $O = \{o_1, \dots, o_n\}$ of simulation outputs. Hence, for a partial model, the fitness value for a test I is determined by the model output yielding the lowest fitness (i.e., the model output revealing the most severe failure or the model output yielding the lowest passable fitness).

Signals First-Order Logic

First-Order Logic of Signals (SFO) [BFHN18b], also known as predicate logic, is a language that defines formal systems using quantified variables over non-logical objects. Unlike propositional logic, it allows the use of sentences that contain variables. First-Order Logic language is used in various domains like mathematics and computer science. The syntax and the semantics of SFO are defined

by typically combining first order logic with linear real arithmetic and uninterpreted unary function symbols, which represent real valued signals over time.

Let $X = \{x_1, x_2, \dots, x_n\}$ be a set of variables, such as $X = T \cup R$ and $T \cap R = \emptyset$, that is the set X is partitioned in two sets T and R containing the *time* and *value* variables, respectively. A *time domain* \mathbb{T} is a non-singular interval of \mathbb{R} . Let \mathbb{T} be a time domain, a *signal* is a function $f : \mathbb{T} \rightarrow \mathbb{R}$. Let $F = \{f_1, f_2, \dots\}$ be a set of signals.

Definition 2 ([\[BFHN18b\]](#)) *A term θ of an SFO formula is either a formula τ or a formula ρ defined according to the grammar below:*

$$\tau \quad ::= \quad t \mid n \mid \tau_1 - \tau_2 \mid \tau_1 + \tau_2 \quad (2.1)$$

$$\rho \quad ::= \quad r \mid f(\tau) \mid n \mid \rho_1 - \rho_2 \mid \rho_1 + \rho_2 \quad (2.2)$$

where $n \in \mathbb{Z}$, $r \in R$, $t \in T$, where R and T are the sets of time and value variables previously defined.

An SFO formula ϕ is defined according to the grammar below:

$$\phi \quad ::= \quad \theta_1 \sim \theta_2 \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \forall r: \phi \mid \forall t \in I: \phi \quad (2.3)$$

where $n \in \mathbb{Z}$, θ_1 and θ_2 are terms, $\sim \in \{<, \leq\}$ and I are intervals with bound in $\mathbb{Z} \cup \{\pm\infty\}$

We use the notation $\forall t' \in t \oplus [a, b]: \phi$ to indicate the following formula $\forall t' \in [-\infty, +\infty]: (a + t \leq t' \wedge t \leq b + t) \rightarrow \phi[t]$; where $\phi[t]$ is a SFO formula whose atoms do not predicate on t .

Given an interpretation of variables and signals, the FOL semantics defines how a FOL formula is interpreted. Let f be a signal, its interpretation w (called trace) is denoted as $\llbracket f \rrbracket_w$. Let x be a variable, its interpretation v (called valuation) is denoted as $\llbracket x \rrbracket_v$.

Definition 3 ([\[BFHN18b\]](#)) *Let w be a trace and v be a valuation, the semantics of a term of an FOL formula, denoted as $\llbracket \theta \rrbracket_{w,v}$, is inductively defined as follows:*

$$\llbracket n \rrbracket_{w,v} = n \text{ for all } n \in \mathbb{Z} \quad (2.4)$$

$$\llbracket \theta_1 - \theta_2 \rrbracket_{w,v} = \llbracket \theta_1 \rrbracket_{w,v} - \llbracket \theta_2 \rrbracket_{w,v} \quad (2.5)$$

$$\llbracket \theta_1 + \theta_2 \rrbracket_{w,v} = \llbracket \theta_1 \rrbracket_{w,v} + \llbracket \theta_2 \rrbracket_{w,v} \quad (2.6)$$

The semantics of an FOL formula, denoted as $(w, v) \models \phi$, is inductively defined as

$$w \models \theta_1 < \theta_2 \quad \text{iff} \quad \llbracket \theta_1 \rrbracket_{w,v} < \llbracket \theta_2 \rrbracket_{w,v} \quad (2.7)$$

$$w \models \forall r: \phi \quad \text{iff} \quad (w, v[r \leftarrow a]) \models \phi \text{ for all } a \in R \quad (2.8)$$

$$w \models \forall t \in I: \phi \quad \text{iff} \quad (w[t \leftarrow a], v) \models \phi \text{ for all } a \in I \quad (2.9)$$

and where boolean connectives are interpreted as usual.

Definition [3](#) indicates, as $v[r \leftarrow a]$, the interpretation v obtained by assigning the value a to the value variable r ; $w[t \leftarrow a]$ indicates the interpretation q obtained by assigning the value a to the time variable t .

It has been shown that the satisfiability of an SFO formula φ is undecidable, while the simpler membership and monitoring problems relative to piecewise-linear traces are decidable. Indeed, monitoring problems can be solved using an algorithm that has polynomial complexity in the size of the input trace and the specification. Specifically, it can be computed in $(m + n)^{2^{O(k+l)}}$, where

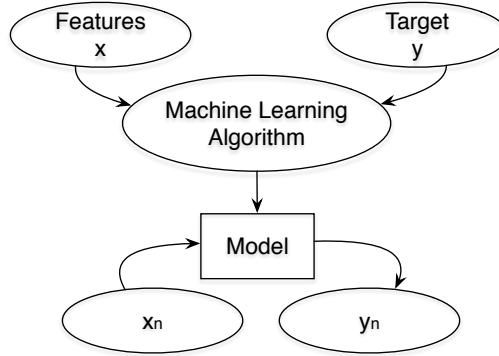


Figure 2.7: Supervised machine learning model: The algorithm learns a model from labeled training data. The model is used to make predictions on the output y_n of the inputs x_n .

m is the length of φ , k is the number of quantifiers in φ , l is the number of function symbols in φ , n is the length of the trace. The algorithm transforms the SFO formula and the trace into a quantified free formula of linear arithmetic by eliminating all the quantifiers, which can be done in time $(m+n)^{2^{k+l}}$. Deciding the validity of the resulting formula can be done in linear time. This complexity is prohibitive for practical applications that have to analyze long traces.

However, it has been shown that the complexity is linear in the size of the trace for a fragment of SFO called bounded-response, in which the intervals $I = [a, b]$ are such that $a \neq \infty$ and $b \neq \infty$. Specifically the monitoring algorithm with complexity $n \cdot 2^{(m+j)2^{O(k+l)}}$ has been proposed, where j is the variability of w relative to φ , and m, k, l and n are defined as previously. The variability of w relative to formula φ is the maximum number of linear segments in w during any time period as long as the horizon of φ .

2.5 Supervised Machine Learning

Predicting the outcome of a given system by learning from the historical data can be done with machine learning. Machine learning is a part of the artificial intelligence where computer algorithms are improved automatically through experience and by the use of data. Supervised machine learning [KZP07] is a machine learning task where we know the outcome of the system under analysis and we can label it. Figure 2.7 shows the supervised machine learning process. Supervised learning automatically learns a function that maps inputs or features to an output based on training input-output pairs. If the output is categorical, the learning task is called classification. If it is numerical, then regression. The ultimate goal of supervised machine learning is to make predictions on the system behaviour.

Supervised machine learning algorithms work by learning structures like trees or by predicting the parameters of a function. The learning process is generally guided by a fitness function that minimizes the difference between the estimated and the predicted output values of a given training data to produce a model which will be used eventually for prediction purposes. Applying machine learning has many advantages. First, the learning model can perform tasks much faster than a human with more reliable results [KK00]. Moreover, GP can be applied in various programming languages. However, one of the major issues of machine learning is its high dependency on training data. Some factors should be then considered when selecting the learning algorithm such as the heterogeneity of the data, the redundancy and the presence of interactions and non-linearities. Some algorithms are easier to apply than others depending on these factors.

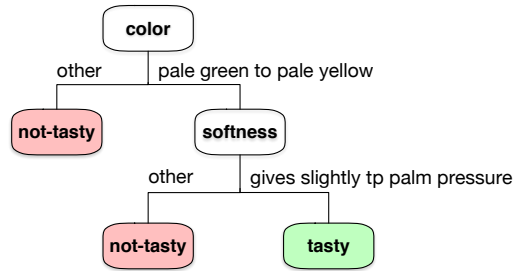


Figure 2.8: An example of decision tree for distinguishing papayas

Decision Trees

Decision trees (DT) [Qui86, RM05, Qui87] algorithms are widely known and applied supervised machine learning algorithms. They provide practical solutions for classification problems (when the data is labeled with a categorical or a qualitative output) and for regression problems (when the data is labeled with continuous or quantitative output).

A decision tree is a hierarchical tree structure that follows the principle of divide-and-conquer strategy to split the source set situated at the root node into the leaves that contain the output mapped to the data. The representation of a decision tree is quite common in the literature [Mur98, SL91]. The internal nodes are labeled with the input features. The arcs coming from each internal node are labeled with a condition on the input feature. These arcs lead to the split of the node into subordinate nodes. Each leaf of the tree is labeled with a class in which the data set has been classified or a probability distribution over the class. Splitting the data set into the optimal subsets typically follows a given splitting rule. Different metrics are used to evaluate and select the best candidate subset. These metrics typically measure homogeneity within the subset where the goal is to generate a node with data subset that all satisfy the condition on the input feature. We provide examples of metrics which provide a measure of the quality of the split:

- Information gain (i.e., C4.5). builds a decision tree using the concept of information entropy. At each node, the attribute that most effectively splits the set of data is selected to make the decision. The splitting criterion is the difference in the entropy where the higher, the better.
- Gini impurity (i.e., CART). applies optimal discriminant analysis and classification tree analysis to find the combination of best splits. These splits are then used to build a statistical model that optimally represents the data.

An example decision tree for checking if a given papaya is tasty or not is shown in Figure 2.8. This decision tree is built using a training set where the attributes are the color of the papaya (ranging from dark green, through orange and red to dark brown) and the softness of the papaya (ranging from rock hard to mushy). In this example, the training set is a sample of papayas that are examined for color and softness. The papayas in the training set are labeled by whether they were tasty or not. To classify the data in the training set, the decision tree algorithm checks the color of the papaya. If the color is in the range *pale to pale yellow*, the algorithm moves to check the softness. The papaya is predicted as *tasty* if it *gives slightly to palm pressure*. Otherwise, the algorithm predicts that the papaya is *not-tasty*.

The main advantage of the decision trees is the interpretability aspect. Decision trees are known for having a simple representation that is easy to understand. Furthermore, they are fast for fitting and prediction, and low on memory usage, but they can have low predictive accuracy. The standard way when applying decision trees is to grow simpler trees to prevent overfitting.

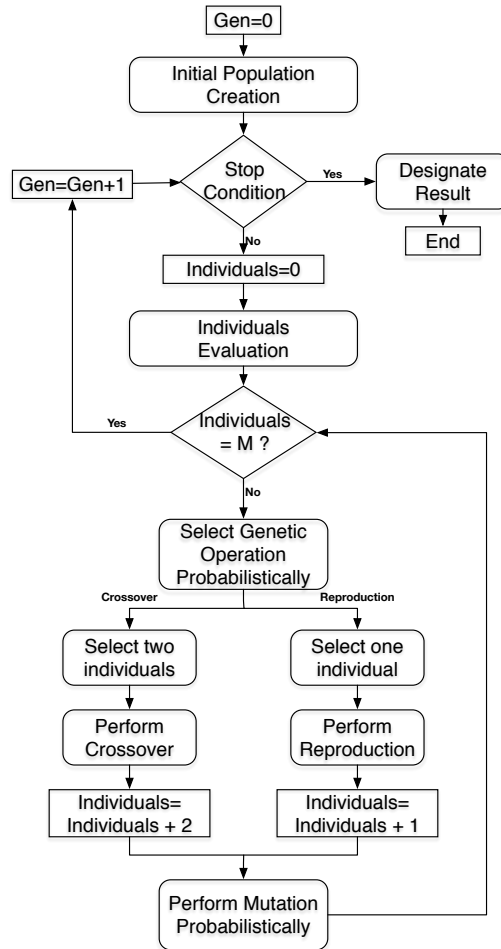


Figure 2.9: Genetic Programming Flowchart

Genetic Programming

Genetic programming (GP) [KK92, PLMK08, BNKF98, GPM20, MAS05] is fundamentally different from other approaches to artificial intelligence, like machine learning, in terms of its representation and its mechanism gleaned from nature. The motivation behind the use of genetic programming is deduced from classical structured programming like object-oriented programming and object libraries, which often leaves the programming task firmly in the hands of the programmer and prevents the freedom of the computer user to manipulate and maintain the code. Enabling computers to learn to program themselves allows to free the computer industry from the explicit code. In fact, when the user is mainly interested in the structure of the program, he just needs to tell the GP exactly how to do and GP would do so. Indeed, GP has been successfully used to evolve software controllers that are more effective than those written by engineers [KK00]. Any system that relies on a population of programs or algorithms that evolve during the search process where new invariants are generated, can be called a genetic programming system. GP has introduced a level of freedom of representation into the machine learning world that did not previously exist. In our work, we focus on GP systems that use (expression) trees to represent programs. Even though the GP structures in the literature may be represented in other ways, research has already confirmed the interpretability of both linear and graph-based genetic programming systems, which is perhaps the most important distinguishing feature of genetic programming. We present our GP technique in Chapter 6.

The idea of Genetic programming is derived from evolutionary biology. The algorithm genetically breeds populations of computer programs to solve problems. In this dissertation, specifically in Chapter 6, we follow the flowchart shown in Figure 2.9. Genetic programming uses four steps to solve problems:

1. Generate an initial population of random trees of the functions and terminals of the problem (computer programs).
2. Execute each program in the population and assign it a fitness value according to how well it solves the problem.
3. Create a new population of computer programs by applying Reproduction, Crossover and Mutation.
4. The best computer program that appeared in any generation is designated as the result of genetic programming.

The genetic programming system needs to be adapted to the problem we aim to solve. Therefore, it is important to answer the following questions that raise contentious issues for the learning algorithm design.

- How are solutions represented in the algorithm? genetic programming allows to represent the solution as any possible computer program. It is theoretically possible for a properly designed GP system to evolve solutions that any other machine learning system could produce. GP indeed supports solutions with boolean, arithmetic and logical operators. These representations are easy to evolve in GP. It also supports the use of conditional branching structures such as IF/THEN statements, strong typing and grammars, which makes it possible for GP to evolve constrained programs.
- Which search operators does the learning algorithm use to move in the solution space? Search operators are the key to a good GP system. Genetic operators not only define and evolve the solutions but they also limit the search space area. Therefore, the choice of the applied operators in GP and configuring rates with appropriate values is crucial. A good GP system would use the suitable operators which select fitted solutions and avoids bad ones. The primary genetic operators applied in GP are the “crossover” and “mutation” operators. Mutation tweaks an existing solution to produce a new one which allows more exploration of the search space. Crossover exchanges parts of two existing solutions to produce two new solutions which allows more exploitation of the discovered solutions. Configuring the rates of these operators in the GP algorithm helps control the exploitation and exploration capabilities of the search depending on the user’s goals.
- How are solutions evaluated? The fitness function serves as a metric to assess which candidate solutions are fitted solutions. The fitter solutions are later exploited by the operators in order to produce more promising solutions for further selection. The fitness function typically limits further the solutions that are generated by genetic operators. Retaining candidate solutions from a population of all possible solutions is recognised as a machine learning method used in evolutionary algorithms.

Chapter 3

Generating Test Oracles for Simulink Models with Continuous and Uncertain Behaviours

Test automation requires automated oracles to assess test outputs. For cyber physical systems (CPS), oracles, in addition to be automated, should ensure some key objectives: (i) they should check test outputs in an online manner to stop expensive test executions as soon as a failure is detected; (ii) they should handle time- and magnitude-continuous CPS behaviors; (iii) they should provide a quantitative degree of satisfaction or failure measure instead of binary pass/fail outputs; and (iv) they should be able to handle uncertainties due to CPS interactions with the environment.

We propose *Simulink Oracles for CPS Requirements with uncertainty (SOCRaTEs)*, an approach for generating online oracles in the form of Simulink blocks based on CPS functional requirements (Section 3.2). SOCRaTEs translates CPS requirements specified in a logic-based language into test oracles specified in Simulink – a widely-used development and simulation language for CPS. Our approach achieves the objectives noted above through the identification of a fragment of Signal First Order logic (SFOL) to specify requirements, the definition of a quantitative semantics for this fragment and a sound translation of the fragment into Simulink.

This chapter highlights the following research contributions:

1. We propose Restricted Signals First-Order Logic (RFOL), a signal-based logic language to specify CPS requirements (Section 3.2). RFOL is a restriction of Signal First Order logic (SFOL) [BFHN18a] that can capture properties of time- and magnitude-continuous signals while enabling the generation of efficient, online test oracles. We define a quantitative semantics for RFOL to compute a measure of fitness for test results as oracle outputs.
2. We develop a procedure to translate RFOL requirements into automated oracles modeled in the Simulink language (Section 3.2). We prove the soundness of our translation with respect to the quantitative semantics of RFOL. Further, we demonstrate that: (1) the generated oracles are able to identify failures as soon as they are revealed (i.e., our oracles are online); and (2) our oracles can handle models containing parameters with uncertain values and signal inputs

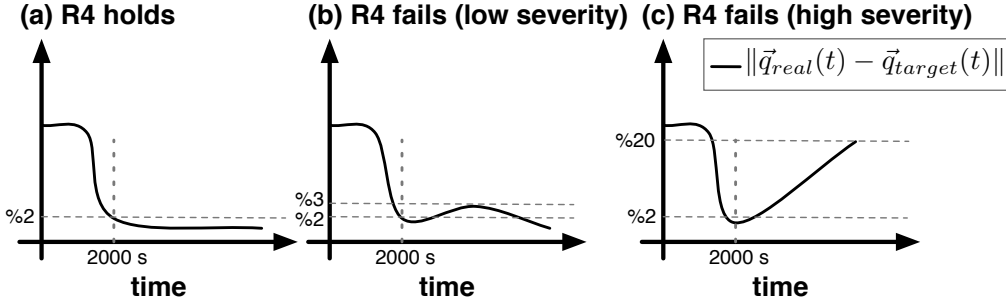


Figure 3.1: Three simulation outputs of our ESAIL case study model indicating the error signal $\|\vec{q}_{real}(t) - \vec{q}_{target}(t)\|$. The signal in (a) passes **R4** in Table 3.1, but those in (b) and (c) violate **R4** with low and high severity, respectively.

with noises by exploiting existing Simulink features. We have implemented our automated oracle generation procedure in a tool which is available online [Soc19].

3. We evaluate our approach using 11 industry Simulink models from two companies in the CPS domain.

Our results show that our proposed logic-based requirements language (RFOL) is sufficiently expressive to specify all the 98 CPS requirements in our industrial case studies. Further, our automated translation can generate online test oracles in Simulink efficiently, and the effort of developing RFOL requirements is acceptable, showing potentials for the practical adoption of our approach. Finally, for large and computationally intensive industry models, our online oracles can bring about dramatic time savings by stopping test executions long before their completion when they find a failure, without imposing a large time overhead when they run together with the model.

Structure. Section 3.1 presents the motivating example and the challenges addressed in this work. Section 3.2 outlines SOCRaTEs and its underlying assumptions, presents the Restricted Signals First-Order Logic and its semantics and describes our automated oracle generation procedure. Section 3.3 evaluates SOCRaTEs and Section 3.5 concludes the chapter.

3.1 Motivation and Challenges

We motivate the work of this chapter using our case study system, the ESAIL maritime micro-satellite. The main functional requirements of ESAIL are presented in the middle column of Table 3.1, and the variables used in the requirements are described in Table 3.2.

Before software coding or generating code from Simulink models (a common practice when Simulink/Matlab models are used), engineers need to ensure that their models satisfy the requirements of interest (e.g., those in Table 3.1). Despite the presence of automated verification tools for Simulink, the verification of industrial CPS Simulink models largely relies on simulation and testing. This is because existing verification tools [RS11, SDV19, REA19] are not amenable to the verification of large Simulink models like ESAIL that contain continuous physical computations and third-party library code [MNBB18, AFS⁺13]. Further, CPS Simulink models often capture dynamic and hybrid systems [Alu15]. It is well-known that model checking such systems is generally undecidable [HKPV98, ACH⁺95, Alu11].

To effectively test CPS models, engineers need to have automated test oracles that can check the correctness of simulation outputs with respect to the requirements. To be effective in the context of CPS testing, oracles should further ensure the following objectives:

Table 3.1: Requirements for the satellite control system (ESAIL) developed by LuxSpace.

ID	Requirement	Restricted Signal First-Order logic formula*
R1	The angular velocity of the satellite shall always be lower than $1.5m/s$.	$\forall t \in [0, 86\,400): \ \vec{w}_{sat}(t)\ < 1.5$
R2	The estimated attitude of the satellite shall be always equal to 1.	$\forall t \in [0, 86\,400): \ \vec{q}_{estimate}(t)\ = 1$
R3	The maximum reaction torque must be equal to $0.015Nm$.	$\forall t \in [0, 86\,400): \ \vec{trq}(t)\ \leq 0.015$
R4	The satellite attitude shall reach close to its target value within 2000 sec (with a deviation not more than 2 degrees) and remain close to its target value.	$\forall t \in [2\,000, 86\,400): \ \vec{q}_{real}(t) - \vec{q}_{target}(t)\ \leq 2$
R5	The satellite target attitude shall not change abruptly: for every t , the difference between the current target attitude and the one at two seconds later shall not be more than α° .	$\forall t \in [0, 86\,400): \ \vec{q}_{target}(t) - \vec{q}_{target}(t+2)\ \leq 2 \times \sin(\frac{\alpha}{2})$
R6	The satellite shall reach close to its desired attitude (with a deviation not more than %2) 2000 sec after it enters its normal mode (i.e., $sm(t) = 1$) and it has stayed in that mode for at least 1 sec.	$\forall t \in [0, 86\,400): (sm(t) = 0 \wedge (\forall t_1 \in (t, t+1]: sm(t_1) = 1) \rightarrow \ \vec{q}_{real}(t+2000) - \vec{q}_{estimate}(t+2000)\ \leq 0.02)$

* The notation \vec{a} indicates that a is a vector; $\|\vec{a}\|$ indicates the norm of the vector.

Table 3.2: Signals variables of the ESAIL model.

Var.	Description	Var.	Description
sm	Satellite mode status.	\vec{trq}	Satellite torque.
\vec{w}_{sat}	Satellite angular velocity.	\vec{q}_{real}	Current satellite attitude.
$\vec{q}_{estimate}$	Estimated satellite attitude.	\vec{q}_{target}	Target satellite attitude.

- *O1. Test oracles should check outputs in an online mode.* An online oracle (a.k.a as a monitor in the literature [BFHN18b]) checks output signals as they are generated by the model under test. Provided with an online oracle, engineers can stop model simulations as soon as failures are identified. Note that CPS Simulink models are often computationally expensive because they have to capture physical systems and processes using high-fidelity mathematical models with continuous behaviors. Further, CPS models have to be executed for a large number of test cases. Also, due to the reactive and dynamic nature of CPS models, individual test executions (i.e., simulations) have to run for long durations to exercise interactions between the system and the environment over time. For example, to simulate the satellite behavior for 24h (i.e., 86 400s), the ESAIL model has to be executed for 84 minutes (~ 1.5 hours) on 12-core Intel Core i7 3.20GHz 32GB of RAM. Further, the 24h-length simulation of ESAIL has to be (re)run for tens or hundreds of test cases. Therefore, online test oracles are instrumental to reduce the total test execution time and to increase the number of executed test cases within a given test budget time.
- *O2. Test oracles should be able to evaluate time and magnitude-continuous signals.* CPS model inputs and outputs are signals, i.e., functions over time. Signals are classified based on their time-domain into time-discrete and time-continuous, and based on their value-range into magnitude-discrete and magnitude-continuous. The type of input and output signals depends on the modeling formalisms. For example, differential equations [New74] often used in physical modeling yield continuous signals, while finite state automata [Hen09] used to specify discrete-event systems generate discrete signals. Figure 3.1 shows three magnitude- and time-continuous signal outputs of ESAIL indicating the error in the satellite attitude,

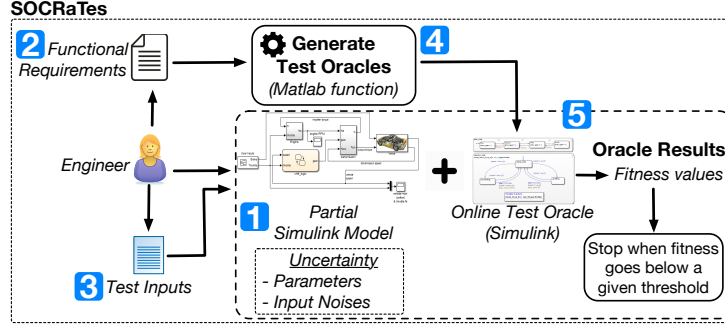


Figure 3.2: Overview of SOCRaTes, our automated oracle generation approach.

i.e., the difference between the real and the target satellite attitudes ($\|\vec{q}_{real}(t) - \vec{q}_{target}(t)\|$). An effective CPS testing framework should be able to handle the input and output signals of different CPS formalisms including the most generic and expressive signal type, i.e., time-continuous and magnitude-continuous. Such testing frameworks are then able to handle any discrete signal as well.

- *O3. Test oracles for CPS should provide a quantitative measure of the degree of satisfaction or violation of a requirement.* Test oracles typically classify test results as failing and passing. The boolean partition into “pass” and “fail”, however, falls short of the practical needs. For CPS, test oracles should assess test results in a more nuanced way to identify among all the passing test cases, those that are more acceptable, and among all the failing test cases, those that reveal more severe failures.

Therefore, an effective test oracle for CPS should assess test results using a *quantitative fitness measure*. For example, the satellite attitude error signal in Figure 3.1(a) satisfies the requirement **R4** in Table 3.1. But, signals in Figures 3.1(b) and (c) violate **R4** since the error signal does not remain below the %2 threshold after 2000s. However, the failure in Figure 3.1(c) is more severe than that in Figure 3.1(b) since the former deviates from the threshold with a larger margin. A quantitative oracle can differentiate between these failures.

- *O4. Test oracles should be able to handle uncertainties in CPS function models.* We consider two main recurring and common sources of uncertainties in CPS [ER14, dWOJ+07]: (1) Uncertainty due to unknown hardware choices which results in model parameters whose values are only known approximately at early design stages. For example, in ESAIL, there are uncertainties in the type of the magnetometer and in the accuracy of the sun sensors mounted on the satellite (see Table 3.3). (2) Uncertainty due to the noise in the inputs received from the environment, particularly in the sensor readings. This is typically captured by white noise signals applied to the model inputs (e.g., Table 3.3 shows the signal-to-noise (S2N) ratios for the magnetometer and sun sensor inputs of ESAIL). Oracles for CPS models should be able to assess outputs of models that contain parameters with uncertain values and signal inputs with noises.

3.2 Approach Overview

Figure 3.2 shows an overview of SOCRaTes (*Simulink Oracles for CPS Requirements with uncertainty*), our approach to generate automated test oracles for CPS models. SOCRaTes takes three inputs: (1) a CPS model with parameters or inputs involving uncertainties, (2) a set of functional

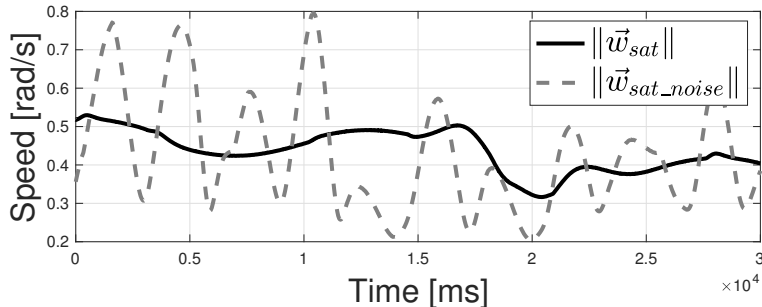


Figure 3.3: Signals $\|\vec{w}_{sat}\|$ for the w_{sat} output of ESAIL. The solid-line signal is generated by ESAIL with no uncertainty, and the dashed-line signal is generated when the S2N ratios in Table 3.3 are applied to the ESAIL inputs.

requirements for the CPS model and (3) a set of test inputs that are developed by engineers to test the CPS model with respect to its requirements.

SOCRaTeS makes the following assumptions about its inputs:

- *A1. The CPS model is described in Simulink (1).* Simulink is a prevalent modeling language in the automotive domain [MNBB16a, ZSM12]. Simulink appeals to engineers since it is particularly suitable for specifying dynamic systems and it allows engineers to test their models as early as possible. We recall from Chapter 2, Section 2.2, the modeling language provided by Simulink and how the system design is constructed and simulated. Specifically, to simulate a Simulink model M , the simulation engine receives signal inputs defined over a time domain and computes signal outputs at successive time steps over the same time domain used for the inputs. For example, Figure 3.3 shows a signal (black solid line) for the w_{sat} output of ESAIL computed over the time domain $[0, 3 \times 10^4]$.
- *A2. Functional requirements are described in a signal logic-based language (2).* We present our requirements language in this section and compare it with existing signal logic languages [BFHN18a, MN04]. We evaluate expressiveness of our language in Section 3.3.
- *A3. A set of test inputs exercising requirements are provided (3).* We assume engineers have a set of test inputs for their CPS model. The test inputs may be generated manually, randomly or based on any test generation framework proposed in the literature [MNBB16a, ZSM12]. Our approach is agnostic to the selected test generation method.

SOCRaTeS automatically converts functional requirements into oracles specified in Simulink (4). The oracles evaluate test outputs of the CPS model in an automated and online manner and generate fitness values that provide engineers with a degree of satisfaction or failure for each test input (5). Engineers can stop running a test in the middle when SOCRaTeS concludes that the test fitness is going to remain below a given threshold for the rest of its execution.

Simulink Models

We recall from Chapter 2, Section 2.2, the modeling language provided by Simulink and the description of CPS Simulink models (1). We note that our oracles rely on Simulink solvers to properly handle signals based on their time domains, whether discrete or continuous. As a result, our work, in contrast to existing techniques, is able to seamlessly handle the verification of logical properties over not just discrete but also continuous CPS models.

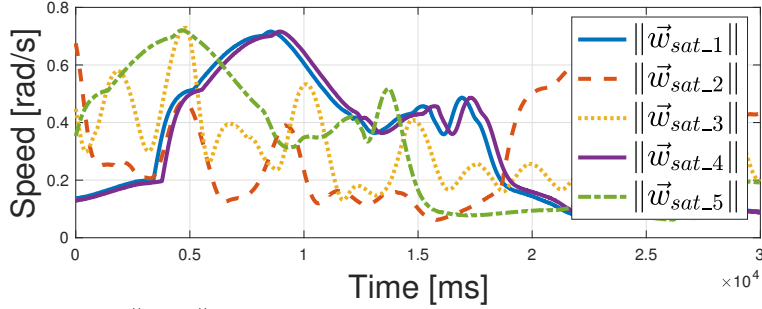


Figure 3.4: A set of signals $\|\vec{w}_{sat}\|$ for the output w_{sat} of ESAIL with uncertain parameters (i.e., when the sun sensor and magnetometer parameters are specified as in Table 3.3).

Table 3.3: Uncertainty in Autopilot: The values of the magnetometer type and the sun sensor accuracy parameters are given as ranges (middle column). The noise values for the magnetometer and sun sensor inputs are given in the right column.

Component	Parameter Values	Noises (S2N)
Magnetometer	[60000, 140000] nT	$100 \cdot e^{-12} \text{ T}/\sqrt{\text{Hz}}$
Sun sensor	$2.9 \cdot 10^{-3} \pm 10\%$	$2.688 \cdot e^{-6} \text{ A}$

Simulink has built-in support to specify and simulate some forms of uncertainty. We refer to Simulink models that contain uncertain elements as *partial* models (denoted M_p), while we use the term *definitive* to indicate models with no uncertainty. Simulink can specifically capture the following two kinds of uncertainty that are common for CPS (O_4):

(i) *Uncertainty due to the noise in inputs.* In Simulink, uncertainty due to the noise is implemented by augmenting model inputs with continuous-time random signals known as *White Noise* (WN) [GK10]. The degree of WN for each input is controlled by a *signal-to-noise ratio* (S2N) value which is the ratio of a desired signal over the background WN [Sig19]. Table 3.3 shows the S2N ratios for two inputs of Autopilot. Fig. 3.3 shows the signal $\|\vec{w}_{sat}(t)\|$ (gray dashed line) after adding some noise to the original w_{sat} signal (black solid line).

(ii) *Uncertainty related to parameters with unknown values.* In Simulink, parameters whose values are uncertain are typically defined using variables of type *uncertain real* (ureal), which is a built-in type in Simulink that specifies a range of values for a variable. Table 3.3 shows two parameters of Autopilot whose exact values are unknown, and hence, value ranges are assigned to them.

Let M_p be a partial Simulink model with n outputs, and let k be the number of different value assignments to uncertain parameters of M_p . A *simulation* of a partial Simulink model M_p , denoted by $H_p(\bar{u}, M_p) = \{\bar{y}_1, \bar{y}_2 \dots \bar{y}_k\}$, receives a set $\bar{u} = \{u_1, u_2 \dots u_m\}$ of input signals defined over the same time domain, and produces a set of simulation outputs $\{\bar{y}_1, \bar{y}_2 \dots \bar{y}_k\}$ such that each \bar{y}_i is generated by one value assignment to uncertain parameters of M_p . Specifically, for each $O_i \in \{\bar{y}_1, \bar{y}_2 \dots \bar{y}_k\}$, we have $O_i = \{y_1, y_2 \dots y_n\}$ such that o_1, \dots, o_n are signals for outputs of M_p , i.e., each O_i contains a signal for each output of M_p . The function H_p generates the simulation outputs consecutively and is provided in the Robust Control Toolbox of Simulink [Rob19] which is the uncertainty modeling and simulation tool of Simulink models with dynamic behavior. The value of k indicating the number of value assignments to uncertain parameters can either be specified by the user or selected based on the recommended settings of H_p . For example, Figure 3.4 plots five simulation outputs for the output w_{sat} of Autopilot. The uncertainty in this figure is due to the sun sensor accuracy parameter that takes values from the range $2.9 \cdot 10^{-3} \pm 10\%$ as indicated in Table 3.3.

Restricted Signals First-Order Logic

We present *Restricted Signals First-Order Logic (RFOL)*, the logic we propose to specify CPS functional requirements (2). Our choice of a language for CPS requirements is mainly driven by the objectives *O1* and *O2*. These two objectives, however, are in conflict. According to *O2*, the language should capture complex properties involving magnitude- and time-continuous signals. Such language is expected to have a high runtime computational complexity [BFHN18b]. This, however, makes the language unsuitable for the description of online oracles that should typically have low runtime computational complexity, thus contradicting *O1*. For example, Signals First Order (SFO) logic [BFHN18b] is an extension of first order logic with continuous signal variables. SFO, however, is not amenable to online checking in its entirety due to its high expressive power that leads to high computational complexity of monitoring SFO properties [BFHN18b]. Thus, the procedure for monitoring SFO properties is tailored to offline checking. In order to achieve objectives *O1* and *O2*, we define Restricted Signals First-Order Logic (RFOL), a fragment of SFO. RFOL can be effectively mapped to Simulink to generate online oracles that run together with the model under test by the same solvers applied to the model, which can handle any signal type (i.e., discrete or continuous), hence addressing objectives *O1* and *O2*. Note that even though RFOL is less expressive than SFO, as we will discuss in Section 3.3, all CPS requirements in our case studies can be captured by RFOL.

RFOL Syntax. Let $T = \{t_1, t_2, \dots, t_d\}$ be a set of *time variables*. Let $F = \{f_1, f_2, \dots, f_l\}$ be a set of signals defined over the same time domain \mathbb{T} , i.e., $f_i : \mathbb{T} \rightarrow \mathbb{R}$ for every $1 \leq i \leq l$.

Let us consider the grammar \mathcal{G} defined as follows:

$$\begin{aligned} \tau &::= t + n \mid t - n \mid t \mid n \\ \rho &::= f(\tau) \mid \mathbf{g}(\rho) \mid \mathbf{h}(\rho_1, \rho_2) \\ \phi &::= \rho \sim r \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \forall t \in \langle \tau_1, \tau_2 \rangle : \phi \mid \exists t \in \langle \tau_1, \tau_2 \rangle : \phi \end{aligned}$$

where $n \in \mathbb{R}_0^+$, $t \in T$, $f \in F$, $r \in \mathbb{R}$, and \mathbf{g} and \mathbf{h} are, respectively, arbitrary unary and binary arithmetic operators, \sim is a relational operator in $\{<, \leq, >, \geq, =, \neq\}$, and $\langle \tau_1, \tau_2 \rangle$ is a *time interval* of \mathbb{T} (i.e., $\langle \tau_1, \tau_2 \rangle \subseteq \mathbb{T}$) with lower bound τ_1 and upper bound τ_2 . The symbols \langle and \rangle are equal to $[$ or $($, respectively to $]$ or $)$, depending on whether τ_1 , respectively τ_2 , are included or excluded from the interval. We refer to τ , ρ and ϕ as *time term*, *signal term* and *formula term*, respectively. A *predicate* is a formula term in the form $\rho \sim r$.

Definition 4 A Restricted Signals First-Order Logic (RFOL) formula φ is a formula term defined according to the grammar \mathcal{G} that also satisfies the following conditions: (1) φ is closed, i.e., it does not have any free variable; and (2) every sub-formula of φ has at most one free time variable.

In RFOL, boolean operations (\wedge , \vee) combine predicates of the form $\rho \sim r$, which compare signal terms with real values¹. The formulae further quantify over time variables of signal terms ρ in $\rho \sim r$ and bound them in time intervals $\langle \tau_1, \tau_2 \rangle$. Table 3.1 shows the formalization of the ESAIL requirements in RFOL. For example, the predicate $\|\vec{w}_{sat}\| < 1.5$ of formula *R1* states that the angular velocity of the satellite should be less than 1.5m/s, and $\forall t \in [0, 86400)$ forces the predicate to hold for a duration of 86400s \simeq 24h, the estimated time required for the satellite to finish an orbit.

RFOL expressiveness. Here, we discuss what types of SFO properties are eliminated from RFOL due to the conditions in Definition 4. Condition 1 in Definition 4 requires closed formulae. RFOL properties must not include free variables (i.e., they should be formulae and not queries) so that

¹Note that in our logic, negation \neg is applied at the level of predicates.

they generate definitive results when checking test outputs. Condition 2 in Definition 4 is needed to ensure that the formulae can be translated into online oracles specified in Simulink. This condition eliminates formulae containing predicates $\rho \sim r$ where ρ includes an arithmetic operator applied to signal segments over different time intervals (i.e., signal segments with different time scopes). For example, the formula $\forall t \in [1, 5] : \forall t' \in [7, 9] : f(t) + f(t') < 4$ is not in RFOL since $f(t) + f(t') < 4$ has two free time variables t and t' (i.e., it violates condition 2 in Definition 4). The predicate $f(t) + f(t') < 4$ in this formula computes the sum of two segments of signal f related to time intervals $[1, 5]$ and $[7, 9]$. Such formulae are excluded from RFOL since during online checking, the operands $f(t)$ and $f(t')$ cannot be simultaneously accessed to compute $f(t) + f(t')$. We note that formulae with arithmetic operators applied to signal segments over the *same* time interval (e.g., R4 and R5 in Table 3.1), or formulae involving different predicates over different time intervals, but connected with *logical* operators (e.g., R6) are included in RFOL.

Comparison with STL. In addition to SFO, Signal Temporal Logic (STL) [MN04] is another logic proposed in the literature that can capture CPS continuous behaviors. We compare RFOL with STL, and in particular, with *bounded* STL since test oracles can only check signals generated up to a given bound. Hence, for our purpose, bounded STL temporal operators have to be applied (e.g., $\mathcal{U}_{[a,b]}$). RFOL subsumes bounded STL since boolean operators of STL can be trivially expressed in RFOL, and any temporal STL formula in the form of $\varphi_1 \mathcal{U}_{[a,b]} \varphi_2$ can also be specified in RFOL using time terms and time intervals. The detailed translation is available online [Soc19].

RFOL Semantics. We propose a (quantitative) semantics for RFOL to help engineers distinguish between different degrees of satisfaction and failure ($O\beta$). As shown in Table 3.1 and also based on RFOL syntax, CPS requirements essentially check predicates $\rho \sim r$ over time. To define a quantitative semantics for RFOL, we need to first define the semantics of these predicates in a quantitative way. In our work, we define a (domain-specific) *diff* function to assign a fitness value to $\rho \sim r$. We require *diff* to have these characteristics: (1) The range of *diff* is $[-1, 1]$. (2) A value in $[0, 1]$ indicates that $\rho \sim r$ holds, and a value in $[-1, 0)$ indicates that $\rho \sim r$ is violated.

Definition 5 *Let diff be a domain-specific semantics function for predicates $\rho \sim r$. Let $F = \{f_1, \dots, f_l\}$ be a set of signals with the same time domain \mathbb{T} . The semantics of an RFOL formula ϕ for the signal set F is denoted by $\llbracket \phi \rrbracket_F$ and is defined as follows:*

$$\begin{aligned}
 \llbracket f(n) \rrbracket_F &= \begin{cases} f(n) & \text{if } f \in F \text{ and } n \in \mathbb{T} \\ \text{undefined} & \text{otherwise} \end{cases} \\
 \llbracket g(\rho) \rrbracket_F &= g(\llbracket \rho \rrbracket_F) \\
 \llbracket h(\rho_1, \rho_2) \rrbracket_F &= h(\llbracket \rho_1 \rrbracket_F, \llbracket \rho_2 \rrbracket_F) \\
 \llbracket \rho \sim r \rrbracket_F &= \text{diff}(\llbracket \rho \rrbracket_F \sim r) \\
 \llbracket \phi_1 \wedge \phi_2 \rrbracket_F &= \min(\llbracket \phi_1 \rrbracket_F, \llbracket \phi_2 \rrbracket_F) \\
 \llbracket \phi_1 \vee \phi_2 \rrbracket_F &= \max(\llbracket \phi_1 \rrbracket_F, \llbracket \phi_2 \rrbracket_F) \\
 \llbracket \forall t \in \langle n_1, n_2 \rangle : \phi \rrbracket_F &= \min_{\forall t' \in \langle n_1, n_2 \rangle} (\llbracket \phi[t \leftarrow t'] \rrbracket_F) \\
 \llbracket \exists t \in \langle n_1, n_2 \rangle : \phi \rrbracket_F &= \max_{\forall t' \in \langle n_1, n_2 \rangle} (\llbracket \phi[t \leftarrow t'] \rrbracket_F)
 \end{aligned}$$

The choice of the *max* and *min* operators for defining the semantics of \exists and \forall is standard [LT88]: the minimum has the same behavior as \wedge and evaluates whether a predicate holds over the entire time interval. Dually, the max operator captures \vee . The semantics of signal terms $f(n)$ depends on whether the signal is included in F and whether n is in the time domain \mathbb{T} , otherwise $f(n)$ is undefined. We say $\varphi \in \text{RFOL}$ is *well-defined with respect to a signal set F* iff no signal term in φ is

undefined. To avoid undefined RFOL formulae, signal time domains \mathbb{T} should be selected such that signal indices are included in \mathbb{T} , and further, the formula should not have negative signal indices. For example, for properties in Table 3.1, we need a time domain $\mathbb{T} = [0, 86400]$ for **R1** to **R4**, a time domain $\mathbb{T} = [0, 86402]$ for **R5**, and a time domain $\mathbb{T} = [0, 88400]$ for **R6**. Finally, we can infer the boolean semantics of RFOL from its quantitative semantics: For every formula term φ , we have $F \models \varphi$ iff $\llbracket \varphi \rrbracket_F \geq 0$. In other words, φ holds over the signal set F iff $\llbracket \varphi \rrbracket_F \geq 0$.

Let $\mu = \llbracket \rho \rrbracket_F - r$. In our work, we define *diff* as follows:

$$\begin{aligned} \text{diff}(\llbracket \rho \rrbracket_F = r) &= \frac{-|\mu|}{|\mu| + 1} & \text{diff}(\llbracket \rho \rrbracket_F \neq r) &= \begin{cases} \frac{|\mu|}{|\mu| + 1} & \text{if } \mu \neq 0 \\ -\epsilon & \text{else} \end{cases} \\ \text{diff}(\llbracket \rho \rrbracket_F \geq r) &= \frac{\mu}{|\mu| + 1} & \text{diff}(\llbracket \rho \rrbracket_F > r) &= \begin{cases} \frac{\mu}{|\mu| + 1} & \text{if } \mu \neq 0 \\ -\epsilon & \text{else} \end{cases} \\ \text{diff}(\llbracket \rho \rrbracket_F \leq r) &= \frac{-\mu}{|\mu| + 1} & \text{diff}(\llbracket \rho \rrbracket_F < r) &= \begin{cases} \frac{-\mu}{|\mu| + 1} & \text{if } \mu \neq 0 \\ -\epsilon & \text{else} \end{cases} \end{aligned}$$

In the above, ϵ is an infinitesimal positive value that ensures *diff* < 0 when $\mu = 0$ and either $<$, $>$ or \neq is used.

Our *diff* function satisfies the two conditions described earlier and is closed under logical \wedge and \vee . For example, $(\rho \leq r) \wedge (\rho \geq r)$ is equal to $(\rho = r)$. Our *diff* function, further, provides a quantitative fitness measure distinguishing between different levels of satisfaction and refutation. Specifically, a higher value of *diff* indicates that $\rho \sim r$ is fitter (i.e., it better satisfies or less severely violates the requirement under analysis). For example, the *diff* value of the predicate $\|\vec{w}_{sat}\| < 1.5$ for the signals shown in Figure 3.4 is above zero implying that the signals satisfy the predicate. In contrast, the *diff* values for signals $\|\vec{q}_{real} - \vec{q}_{target}\|$ in Figures 3.1(b) and (c) are -0.5 and -0.95 , respectively. This shows that the violation in Figure 3.1(c) is more severe than that in Figure 3.1(b).

The above *diff* function is only one alternative where we assume the fitness is proportional to the difference between ρ and r . We can define the *diff* function differently as long as the two properties described earlier are respected and the proposed semantics for *diff* respects logical conjunction and disjunction operators.

Oracles Generation

We present the oracle generation component of SOCRaTes (4 in Figure 3.2). This component automatically translates RFOL formulae into *online* test oracles specified in Simulink that can handle time- and magnitude-continuous signals and conform to our notion of oracle (Definition 1 introduced in Chapter 2), Section 2.4. Note that an RFOL formula may not be *directly* translatable into an online test oracle if it contains sub-formulae referring to future time instants or to signal values that are not yet generated at the current simulation time. For example, consider the predicate $\|\vec{q}_{real}(t + 2000) - \vec{q}_{estimate}(t + 2000)\| \leq 0.02$ in the **R6** property of Table 3.1. The fitness value of this predicate at t (i.e., the oracle output in Definition 1) can only be evaluated after generating signals \vec{q}_{real} and $\vec{q}_{estimate}$ up to the time instant $t + 2000$. This requires extending the time domain \mathbb{T} by 2000 seconds. Instead of forcing a longer simulation time, we propose a procedure that rewrites the RFOL formulae into a form that allows a direct translation into online test oracles. Having applied the *time and interval shifting* procedure to RFOL formulae, we describe our translation to convert RFOL formulae into *Simulink oracles*. We further present a proof of soundness and completeness of our translation in this section. All the proofs of the Theorems are provided in our online Appendix Soc19. Below, we present the *time-* and *interval-shifting* steps:

Time-shifting. Any signal term that refers to a signal value generated in the future should be rewritten as a signal term that does not refer to the future. For example, the formula $\bar{q}_{real}(t + 2000) < 5$ that refers to the value of \bar{q}_{real} in the future cannot be checked online. Therefore, our time-shifting procedure replaces any signal term $f(t + n)$ with a signal term $f(t - n)$ as follows: Let ψ be an RFOL formula. We traverse ψ from its leaves to its root and replace every sub-formula $\forall t \in \langle n_1, n_2 \rangle : \phi(t)$ (resp. $\exists t \in \langle n_1, n_2 \rangle : \phi(t)$) of ψ with $\forall t \in \langle n_1 + d_t, n_2 + d_t \rangle : \phi(t - d_t)$ (resp. $\exists t \in \langle n_1 + d_t, n_2 + d_t \rangle : \phi(t - d_t)$), where d_t is the maximum value of constant n in time terms $t + n$ appearing as signal indices in $\phi(t)$. For example, the requirement **R5** in Table 3.1 is rewritten as:

$$\forall t \in [2, 86\ 402] : \|\bar{q}_{target}(t - 2) - \bar{q}_{target}(t)\| \leq 2 \times \sin(\frac{\alpha}{2})$$

Interval-Shifting. To ensure that ψ can be translated into an online test oracle, for any $\forall t \in \langle \tau_1, \tau_2 \rangle : \phi$ in ψ , the interval $\langle \tau_1, \tau_2 \rangle$ should end after all the intervals $\langle \tau'_1, \tau'_2 \rangle$ such that $\forall t' \in \langle \tau'_1, \tau'_2 \rangle : \phi'$ is a sub-formula of ϕ (i.e., $\tau_2 \geq \tau'_2$), and further, it should begin after all the intervals $\langle \tau'_1, \tau'_2 \rangle$ such that $\exists t' \in \langle \tau'_1, \tau'_2 \rangle : \phi'$ is a sub-formula of ϕ (i.e., $\tau_1 \geq \tau'_2$). Similarly, for any $\exists t \in \langle \tau_1, \tau_2 \rangle : \phi$ in ψ , the dual of the above two conditions must hold. These conditions will ensure that the evaluation of the sub-formulae in the scope of t can be fully contained and completed within the evaluation of their outer formula. For example, $\forall t \in [0, 3] : (f(t) = 0 \wedge \forall t' \in [0, 5] : f(t') = 1)$ cannot be checked in an online way since the time interval of the inner sub-formula (i.e., $[0, 5]$) does not end before the time interval of the outer formula (i.e., $[0, 3]$). Therefore, our interval-shifting procedure shifts each time interval $\langle \tau_1, \tau_2 \rangle$ to ensure that it terminates after all its related inner time intervals.

Let ψ be an RFOL formula. We traverse ψ from its leaves to its root and we perform the following operations: (i) replace every sub-formula $\forall t \in \langle \tau_1, \tau_2 \rangle : \phi(t)$ of ψ with $\forall t \in \langle \tau_1 + d_u, \tau_2 + d_u \rangle : \phi(t - d_u)$, where d_u is the maximum value of constant n in the upper bounds τ_2 of time intervals $\langle \tau_1, \tau_2 \rangle$ associated with \forall operators and the lower bounds τ_1 of time intervals $\langle \tau_1, \tau_2 \rangle$ associated with \exists operators in $\phi(t)$; (ii) execute a dual procedure to update the time intervals of existential sub-formulae. For example, the interval-shifting procedure rewrites the formula previously introduced as $\forall t \in [2, 5] : (f(t - 2) = 0 \wedge \forall t' \in [0, 5] : f(t') = 1)$.

To ensure interval-shifting is applied to signal variables with constant indices, we replace every $f(n)$ in ψ where n is a constant with $\forall t^* \in [n, n] : f(t^*)$ where t^* is a new time variable that has not been used in ψ . We refer to the RFOL formula obtained by sequentially applying time-shifting and interval-shifting to an RFOL formula φ as *shifted-formula* and denote it by φ_{\uparrow} .

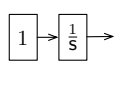


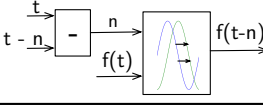
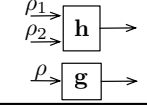
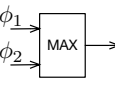
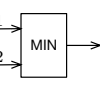
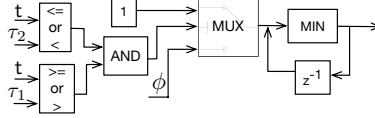
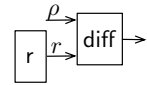
Theorem 1 *Let φ be an RFOL formula and let φ_{\uparrow} be its shifted-formula. For any signal set F , we have: $\llbracket \varphi \rrbracket_F = \llbracket \varphi_{\uparrow} \rrbracket_F$*

The time complexity of generating a *shifted-formula* φ_{\uparrow} is $|\varphi|$ where $|\varphi|$ is the size of the formula φ , i.e., the sum of the number of its temporal and arithmetic operators. Both the time and the interval shiftings scan the syntax tree of φ from its leaves to the root twice: one for computing the shifting values d_t and d_u for every subformula of φ ; and the other to apply the shifting, i.e., replacing the variable t with $t - d_t$ or $t - d_u$.

We translated RFOL formulae written in their shifted-forms into Simulink. Table 3.4 presents the rules for translating each syntactic construct of RFOL defined in Definition 4 into Simulink blocks. Note that **h** and **g** in Table 3.4, respectively, refer to binary arithmetic operators (e.g., $+$) or unary functions (e.g., \sin) and map to their corresponding Simulink operations. Below, we discuss the rules for t , $f(t - n)$, $\forall t \in \langle \tau_1, \tau_2 \rangle : \phi$, and $\rho \sim r$ since the other rules in Table 3.4 directly follow from the RFOL semantics. Note that signal variables in shifted formulae are all written as $f(t - n)$ s.t. $n \geq 0$. Hence, we give a translation rule for signal variables in the form of $f(t - n)$ only.

- Rule1: To compute the value of t , we use an integrator Simulink block to compute the formula $\int_0^t dt$ which yields t .

Table 3.4: Translating the SFFO formulae into Simulink Oracles.

Rule	Rule1	Rule2	Rule3	Rule4	Rule5
Formula	t	n	$t \pm n$	$f(t - n)$	$\mathbf{h}(\rho_1, \rho_2)/\mathbf{g}(\rho)$
Simulink					
Rule	Rule6	Rule7	Rule8		Rule9
Formula	$\phi_1 \vee \phi_2$	$\phi_1 \wedge \phi_2$	$\forall t \in \langle \tau_1, \tau_2 \rangle : \phi$		$\rho \sim r$
Simulink					

- Rule4: To encode $f(t - n)$, we first obtain the delay n applied to the signal f . To obtain the value of n from $t - n$, we compute $t - (t - n)$. We then use the *transport delay* block of Simulink to obtain the value of f at n time instants before t .
- Rule8: The formula $\forall t \in \langle \tau_1, \tau_2 \rangle : \phi$ is mapped into a Simulink model that initially generates the value 1 until the start of the time interval $\langle \tau_1, \tau_2 \rangle$. When $t \in \langle \tau_1, \tau_2 \rangle$ holds, the multiplexer of Rule8 selects the value of ϕ instead of 1. Note that we use symbol $<$ for $\langle = "("$, symbol \leq for $\langle = "["$, symbol $>$ for $\rangle = ")"$, and symbol \geq for $\rangle = "]"$. The feedback loop in the model combined with a delay block (i.e., z^{-1}) computes the minimum of ϕ over the time interval $\langle \tau_1, \tau_2 \rangle$. Once the time interval $\langle \tau_1, \tau_2 \rangle$ expires, the multiplexer chooses constant 1 again. This, however, has no side-effect on the value v already computed for the formula $\forall t \in \langle \tau_1, \tau_2 \rangle : \phi$ because $v \leq 1$ and the minimum of v and 1 remains v until the end of the simulation. Note that the rule for translating $\exists t \in \langle \tau_1, \tau_2 \rangle : \phi$ into Simulink is simply obtained by replacing in Rule8 MIN with MAX and constant 1 with constant -1.
- Rule9: Recall that the semantics of $\rho \sim r$ depends on a domain specific fitness function. In our work, we implement the diff block in Rule9 based on the functions described earlier for function *diff*.

Let φ be an RFOF formula and φ_{\uparrow} be its corresponding shifted formula. We denote by M_{φ} the Simulink model obtained by translating φ_{\uparrow} using the rules in Table 3.4. The model M_{φ} is a definitive Simulink model and has one and only one output because every model fragment in Table 3.4 has one single output. This output will be indicated in the following with the symbol e . Below, we argue that M_{φ} conforms to our notion of test oracle given in Definition 1, and is an online oracle that can handle continuous signals. In order to use M_{φ} to check outputs of model M_p with respect to a property φ , it suffices to connect the outputs of M_p to the inputs of M_{φ} . We denote the model obtained by connecting the output ports of M_p to the input ports of M_{φ} by $M_p + M_{\varphi}$. Clearly, $M_p + M_{\varphi}$ has only one output signal e (i.e., the output of M_{φ}).

Theorem 2 *Let M_p be a (partial) Simulink model, and let I be a test input for M_p defined over the time domain $\mathbb{T} = [0, t_u]$. Let φ be a requirement of M_p in RFOF. Suppose $\{\bar{y}_1, \bar{y}_2 \dots \bar{y}_k\} = H_p(\bar{u}, M_p)$ and $\{\{e_1\}, \{e_2\}, \dots, \{e_k\}\} = H_p(I, M_p + M_{\varphi})$ are simulation results generated for the time domain \mathbb{T} . Then, the value of φ over every signal set $O_i \in \{\bar{y}_1, \bar{y}_2 \dots \bar{y}_k\}$ is equal to the value of the signal e_i generated by $M_p + M_{\varphi}$ at time t_u . That is, $\llbracket \varphi \rrbracket_{O_i} = e_i(t_u)$. Further, we have:*

$$\text{oracle}(M_p, I, \varphi) = \min_{e \in \{e_1, \dots, e_k\}} e(t_u)$$

That is, the minimum value of the outputs of $M_p + M_{\varphi}$ at t_u is equal to the oracle value as defined by Definition 1.

Theorem 4.2 states that our translation of RFOL formulae into Simulink is *sound* and *complete* with respect to our notion of oracle in Definition 1. Note that in the case of a definite Simulink model M , the output of $M + M_\varphi$ is a single signal e . In summary, according to Theorem 4.2, $M_p + M_\varphi$ (or $M + M_\varphi$) is able to correctly compute the fitness value of φ for test input I .

Theorem 3 *Let M_p be a (partial) Simulink model, and let I be a test input for M_p over the time domain $\mathbb{T} = [0, t_u]$. Let φ be a requirement of M_p in RFOL. Suppose $\{\{e_1\}, \{e_2\}, \dots, \{e_k\}\} = H_p(I, M_p + M_\varphi)$ are simulation results generated for \mathbb{T} . Let d be the maximum constant appearing in the upper bounds of the time intervals of φ_\uparrow for existential quantifiers (i.e., time intervals in the form of $\exists t \in [\tau_1, \tau_2] : \phi$ in φ_\uparrow). Each $e_i \in \{\{e_1\}, \{e_2\}, \dots, \{e_k\}\}$ is decreasing over the time interval $(d, t_u]$.*

Note that d in Theorem 4.3 indicates the time instant when all the existentially quantified time intervals of φ are terminated, and hence all the sub-formulae within the existential quantifiers of φ are evaluated. According to Theorem 4.3, the oracle output for φ becomes monotonically decreasing after d . Therefore, after d , we can stop model simulations as soon as the output of $M_p + M_\varphi$ falls below some desired threshold level. More specifically, if the output of $M_p + M_\varphi$ falls below a threshold at time $t > d$ it will remain below that threshold for any $t' \geq t$. Hence, $M_p + M_\varphi$ is able to check test outputs in an online manner and stop simulations within the time interval $(d, t_u]$ as soon as some undesired results are detected. Note that $d = 0$ if φ does not have any existential quantifier.

Our oracles can check Simulink models with time and magnitude-continuous signal outputs since all the blocks used in Table 3.4 can be executed by both fixed-step and variable-step solvers of Simulink, where the time step is decided by the same solver applied to the model under test. Finally, the running time of our oracle is linear in the size of the underlying time domain \mathbb{T} .

3.3 Evaluation

In this section, we empirically evaluate SOCRaTEs using eleven realistic and industrial Simulink models from the CPS domain. Specifically, we aim to answer the following questions. **RQ1:** Is our requirements language (RFOL) able to capture CPS requirements in industrial settings? **RQ2:** Is the use of RFOL and our proposed translation into Simulink models likely to be practical and beneficial? **RQ3:** Is a significant amount of execution time saved when using online test oracles, as compared to offline checking?

Implementation. We implemented SOCRaTEs as an Eclipse plugin using Xtext [Xte19] and Sirius [sir19], and have made it available online [Soc19].

Study Subjects. We evaluate our approach using eleven case studies listed in Table 3.5. We received the case studies from two industry partners: LuxSpace, a satellite system developer, and QRA Corp, a verification tool vendor to the aerospace, automotive and defense sectors. Each case study includes a Simulink model and a set of functional requirements in natural language that must be satisfied by the model. Two of our case studies, i.e., ESAIL from LuxSpace and Autopilot, a public-domain benchmark of Simulink models provided by Lockheed Martin [loc20], are large-scale industrial models and respectively represent full behaviors of a satellite and an autopilot system and their environment. The other nine models capture smaller systems or sub-systems of some CPS. Our case study models implement diverse CPS functions and capture complex behaviors such as non-linear and differential equations, continuous behaviors and uncertainty. ESAIL and Autopilot are continuous models. ESAIL further has inputs with noise and some parameters with uncertain values.

Table 3.5: Important characteristics of our case study systems (from left to right): (1) name, (2) description, (3) number of blocks of the Simulink model of each case study (**#Blocks**), (4) number of requirements in each case study (**#Reqs**) and (5) total number of blocks necessary to encode the requirements (**#BIReqs**)

Model Name	Model Description	#Blocks	#Reqs	#BIReqs
Autopilot	A full six degree of freedom simulation of a single-engined high-wing propeller-driven airplane with autopilot.	1549	12	978
ESAIL	Discussed in Section 3.1	2192	8	292
Neural Network	A two-input single-output predictor neural network model with two hidden layers.	704	6	131
Tustin	A numeric model that computes integral over time.	57	5	463
Regulator	A typical PID controller.	308	10	300
Nonlinear Guidance	A non-linear guidance algorithm for an Unmanned Aerial Vehicles (UAV) to follow a moving target.	373	1	186
System Wide Integrity Monitor	A numerical algorithm that computes warning to an operator when the airspeed is approaching a boundary where an evasive fly up maneuver cannot be achieved.	164	3	169
Effector Blender	A control allocation method to calculate the optimal effector configuration for a vehicle.	95	3	391
Two Tanks	A two tanks system where a controller regulates the incoming and outgoing flows of the tanks.	498	31	1791
Finite State Machine	A finite state machine executing in real-time that turn on the autopilot mode in case of some environment hazard.	303	13	748
Euler	A mathematical model to compute 3-dimensional rotation matrices for an Inertial frame in a Euclidean space.	834	8	834

Table 3.5 also reports the number of blocks (**#Blocks**) of the Simulink models and the number of requirements (**#Reqs**) in our case studies. In total, our case studies include 98 requirements.

RQ1 (RFOL expressiveness). To answer this question, we manually formulated the 98 functional requirements in our case studies into the RFOL language. All of the 98 functional requirements of our eleven study subjects were expressible in RFOL without any need to alter or restrict the requirements descriptions. Further, all the syntactic constructs of RFOL described in this chapter were needed to express the requirements in our study.

The answer to RQ1 is that RFOL is sufficiently expressive to capture all the 98 CPS requirements of our industrial case studies.

RQ2 (Usefulness of the translation). Recall that engineers need to write requirements in RFOL before they can translate them into Simulink. To answer this question, we report the size of RFOL formulas used as input to our approach, the time it takes to generate online Simulink oracles and the size of the generated Simulink oracles. We measure the size of RFOL requirements as the sum of the number of quantifiers, the arithmetic and logical operators, and the size of Simulink oracles as their number of blocks and connections. Figure 3.5(a) shows the size of RFOL formulas ($|\varphi|$) for our case study requirements, and Figure 3.5(b) shows the number of blocks (**#Blocks**) and connections (**#Connections**) of the oracle Simulink models that are automatically generated by our approach. In addition, Figure 3.5(c) shows the time taken by our approach to generate oracle models from RFOL formulas. The total number of blocks necessary to encode all the requirements for each case study is reported in Table 3.5. As shown in Figure 3.5, it took on average 1.6ms to automatically generate oracle models with an average number of 64.2 blocks and 72.6 connections for our 98 case study requirements. Further, the average size of RFOL formulas is 19.2, showing that the pre-requisite effort to write the input RFOL formulas for our approach is not high. The difference in size between RFOL formulas and their corresponding Simulink models is mostly due to

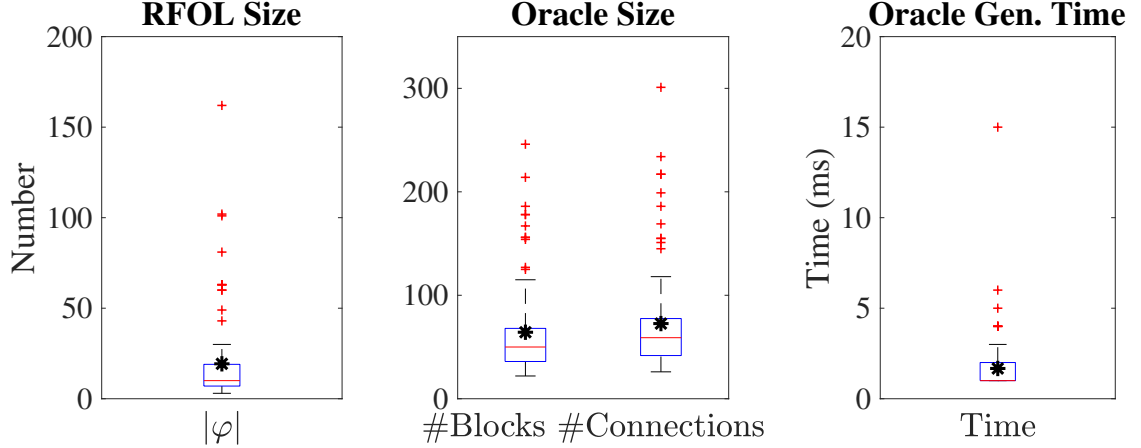


Figure 3.5: Plots reporting (a) the size of the RFOL formulas, (b) the number of blocks and connections of the oracle models and (c) the time took SOCRaTEs to generate the oracles.

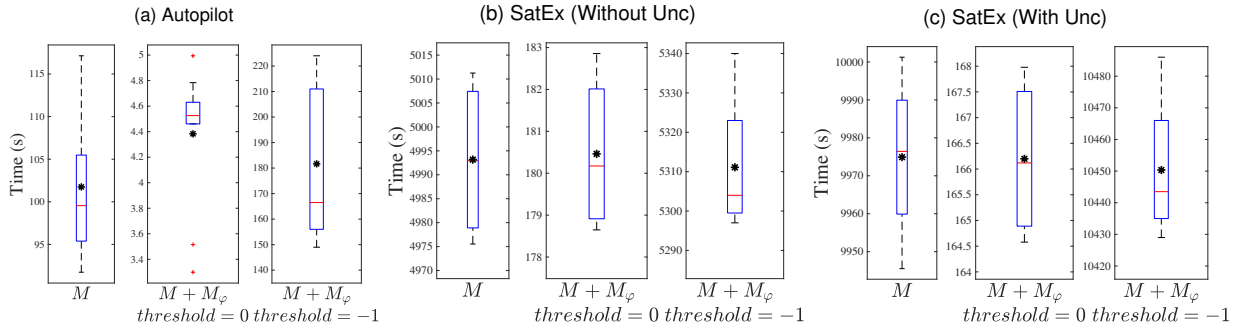


Figure 3.6: Test execution time on models without oracles (M), models with oracles ($M + M_\varphi$) with **threshold** = 0 and models with oracles ($M + M_\varphi$) with **threshold** = -1 for (a) Autopilot, (b) ESAIL without uncertainty and (c) ESAIL with uncertainty.

the former being particularly suitable for expressing declarative properties, such as logical properties with several nested quantifiers. Given this property, and in addition the fact that verification and test engineers are not always very familiar with Simulink — a tool dedicated to control engineers, we expect significant benefits from translating RFOL into Simulink.

The answer to RQ2 is that, for our industrial case studies, the translation into Simulink models is practical as the time required to generate the oracles is acceptable. It takes on average 1.6ms for SOCRaTEs to generate oracle models, and the average size of the input RFOL formulas is 19.2, showing that the pre-requisite effort of our approach is manageable.

RQ3 (Impact on the execution time). Online oracles can save time by stopping test executions before their completion when they find a failure. However, by combining a model M and a test oracle (i.e., generating $M + M_\varphi$), the model size increases, and so does its execution time. Hence, in RQ3, we compare the time saved by online oracles versus the time overhead of running the oracles together with the models. For this question, we focus on our two large industrial models, ESAIL and Autopilot, since they have long and time-consuming simulations while the other models in Table 6.4 are relatively small with simulation times less than one minute. For such models, both the time savings and the time overheads of our online oracles are practically insignificant.

During their internal testing, our partners identified some faults in ESAIL and Autopilot violating some of the model requirements. We received, from our partners, 10 failing test inputs for Autopilot defined over the time domain $\mathbb{T} = [0, 4000]$, and 4 failing test inputs for ESAIL defined over the

time domain $\mathbb{T} = [0, 86400]$. Recall that ESAIL contains some parameters with uncertain values. We also received the value range for one uncertain parameter of ESAIL, i.e., `ACM_type`, from our partner. We then performed the following three experiments. **EXPI**: We ran all the test inputs on the models alone without including oracle models. **EXPII**: We combined ESAIL and Autopilot with all the test oracle models related to their respective requirements and ran all the test inputs on the models with oracles. We did not consider any uncertainty in ESAIL and set `ACM_type` to a fixed value. **EXPIII**: We ran all the tests on ESAIL combined with its oracle models related to its requirements and defined `ACM_type` as an uncertain parameter with a value range. We repeated **EXPII** and **EXPIII** for two threshold values: $threshold = 0$ where test executions are stopped when tests fail according to their boolean semantics, and $threshold = -1$ where test executions are never stopped.

Figures 3.6(a) and (b), respectively, show the results of **EXPI** and **EXPII** for Autopilot and ESAIL. Note that box plots have different scales. Specifically, the figures show the time required to run the test inputs on Autopilot and ESAIL (1) without any oracle model (M), (2) with oracle models ($M + M_\varphi$) for $threshold = 0$, and (3) with oracle models ($M + M_\varphi$) for $threshold = -1$. Specifically, in the second case, test oracles stop test executions when test cases fail, and in the third case, test oracles are executed together with the model, but do not stop test executions. Our results show that on average it takes 101.1s and 4993.2s to run tests on Autopilot and ESAIL, respectively (i.e., Case M). These averages, respectively, reduce to 4.3s and 180.4s when oracles stop test executions, and they, respectively, increase to 181.6s and 5311.1s when oracles do not stop test executions. That is, for Autopilot, the average time saving of our oracles is 95.6% ($\approx 1.5m$) while their average time overhead is 78% ($\approx 1.2m$). In contrast, for ESAIL, our oracles lead to an average time saving of 96% ($\approx 80m$) and an average time overhead of 6% ($\approx 5m$). We note that Autopilot is less computationally intensive. In this case, the time savings and overheads are almost equivalent because the size and complexity of the generated oracles are comparable to those of the model. ESAIL, on the other hand, is more computationally intensive, and as the results show, for ESAIL our oracles introduce very little time overhead but are able to save a great deal of time when they identify failures. Finally, we note that the time saving depends also on the presence of faults in models and whether and when test cases trigger failures. Nevertheless, according to discussions with our partners, and as evidenced by our case studies, early CPS Simulink models typically contain faults, and hence, our approach can help in saving test execution times for such models.

Figure 3.6(c) shows the results of **EXPIII** for running ESAIL with uncertainty. Since in the case of uncertainty, a set of outputs are generated, the total test execution time increases. Specifically, it takes, on average, 9974.9s to run ESAIL with uncertainty without oracles, 166.2s to run it when oracles stop test executions, and 10450.0s to run it when oracles do not stop test executions. As the results show, for ESAIL with uncertainty, the time saving is even higher (i.e., 98%, $\approx 163m$) than the case of ESAIL without uncertainty, because oracles stop simulations as soon as one output among the set of generated outputs fails.

The answer to RQ3 is that, for large and computationally intensive industrial models, our oracles introduce very little time overhead (6%) but are able to save a great deal of time when they identify failures (96%). When models contain uncertainty the time saving becomes even larger and the time overhead decreases, making our online oracles more beneficial.

Data Availability. Our data and tool are available online [Soc19] and are also submitted alongside the paper of the work presented in this Chapter [MNGB19]. Among the considered models, all the models with the exception of the SatEx model are made available. The SatEx model is not shared as it is part of a non-disclosure agreement.

3.4 Related Works

In this section, we provide the related work in the context of oracle generation. Several research approaches have been proposed to address the oracle problem in Software testing [DHF14, JBG16, FP09, DM10, MN04, BDN17, NDB13a, BFHN18a, SNBB18, MN13, BDNCT17, AMN12, DR96, SBB+18, PW18, HMF15, DY94, MMM95, CHS05, LGA96, SL02, SBLR07, MMS96, SC93, RAO92, LH01, WSB+09, MBG+20b].

According to a recent survey [BDD+18], all the research threads described earlier provide a set of inputs exercising requirements. However, only few of these approaches [DHF14, JBG16, FP09, DM10, MN04, BDN17, NDB13a, BFHN18a, SNBB18, MN13, BDNCT17, AMN12, DR96, SBB+18, AAI+21] build on CPS models described in Simulink and use signal logic-based language to express their functional requirements.

Dokhanchi et al. [DHF14] propose an online monitoring procedure for Metric Temporal Logic (MTL) [Koy90a] properties implemented in the S-TaLiRo tool [ALFS11b]. The work builds on CPS models described in Simulink and uses a set of functional requirements described in a signal logic-based language. The authors use a prediction technique to handle temporal operators that refer to future time instants compared to the shifting procedures proposed in our work. As a result, their monitoring procedure has a higher running time complexity than our oracles (i.e., polynomial in the size of time history versus linear in the time domain \mathbb{T} size). Furthermore, they do not translate their monitors into Simulink, and hence, cannot benefit from the execution time speed-up of efficient Simulink blocks and the Simulink variable step solvers to handle continuous behaviors. Thus, as shown by Dokhanchi et al. [DHF14], the time overhead of their approach is considerably high as the time history grows. Jakvsic et al. [JBG16] recently developed an online monitoring procedure for STL by translating STL into automata monitors with a complexity that is exponential in the size of the formula. In contrast to our work, such monitors are not able to handle continuous signals sampled at a variable rate directly. This such signals are approximated as fixed-step signals, hence decreasing the analysis precision of continuous behaviors. To the best of our knowledge and according to a recent survey [BDD+18], the only work that, like us, translates a logic into Simulink to enable online monitoring is the work of Balsini et al. [BDNCT17]. The translation, however, is given for a restricted version of STL, which for example does not allow the nesting of more than two temporal operators. As discussed in Section 3.2, RFOL subsumes STL. Hence, our translation subsumes that of Balsini et al. [BDNCT17]. Breach [Don10, DM10] is a monitoring framework for continuous and hybrid systems that translates STL into online monitors specified in C++ or MATLAB S-functions. However, due to the overhead of integrating C++ or S-functions in Simulink, running monitors in the Breach framework greatly slows down model simulations, by 4.5 times [WKLS18], making the monitors impractical for computationally expensive CPS models such as our Autopilot case study. Finally, Maler et al. [MN13] propose a monitoring procedure that receives signal segments sequentially, checks each segment and stops simulations if a failure is detected. This work, however, is only partially online since each segment is eventually checked in an offline mode.

3.5 Conclusions

In this chapter, we presented SOCRaTes, an automated approach to generate online test oracles in Simulink able to handle CPS Simulink models with continuous behaviors and involving uncertainties. Our oracles generate a quantitative degree of satisfaction or failure for each test input. Our results were obtained by applying SOCRaTes to 11 industry case studies and show that (i) our requirements language is able to express all the 98 requirements of our case studies; (ii) the effort required by SOCRaTes to generate online oracles in Simulink is acceptable, paving the way for a practical

adoption of the approach, and (iii) for large models, our approach dramatically reduces the test execution time compared to when test outputs are checked in an offline manner. In the next chapter, we use our industrial benchmark to empirically compare two mainstream verification techniques, namely, *model testing* and *model checking*.

Chapter 4

Evaluating Model Testing and Model Checking for Finding Requirements Violations in Simulink Models

In this chapter, we report on an empirical study to evaluate capabilities of model testing and model checking techniques in finding faults in Simulink models. In our empirical study, we use a benchmark consisting of Simulink models from the CPS industry to compare those methods. The benchmark includes eleven public-domain Simulink models provided by Lockheed Martin [loc20]. The Simulink models are representative of different types of CPS behavioral models in the aerospace and defense sector. Each model is accompanied by a set of functional requirements described in natural language that must be satisfied by the model. Each model further includes some faults that violate some of the model requirements. Model checking tool vendors generally use such representative benchmarks to assess the capabilities of different verification and testing tools in the market. these companies are requested to identify as many requirements violations as possible when provided with the benchmark.

In this chapter, the model testing technique builds on prior approaches in this area [MNBB18]. Our proposed technique, shown in Figure 4.1(a), is implemented as a typical search-based testing framework [McM04]. In this framework, meta-heuristic search algorithms [Luk13] are used to explore the test input space and to select the best test inputs, i.e., the test inputs that reveal or are close to revealing requirements violations. The search is typically guided by a fitness function that acts as a distance function and estimates how far test inputs are from violating a certain requirement. In this chapter, we use a search algorithm based on Hill Climbing heuristic [Luk13] that, in prior work [MNB⁺15b], has shown to be effective in testing Simulink models. We define search fitness functions using existing translations of logical formulas into quantitative functions estimating degrees of satisfaction of the formulas [AFS⁺13].

This chapter highlights the following main contributions:

1. We provide a categorization of CPS model types and a set of common logical patterns in CPS functional requirements. Using our industrial benchmark Simulink models, we identify a categorization of the CPS models based on their functions. We further formalize the textual requirements in a logic-based requirements language and identify some common patterns among the CPS requirements in the benchmark.

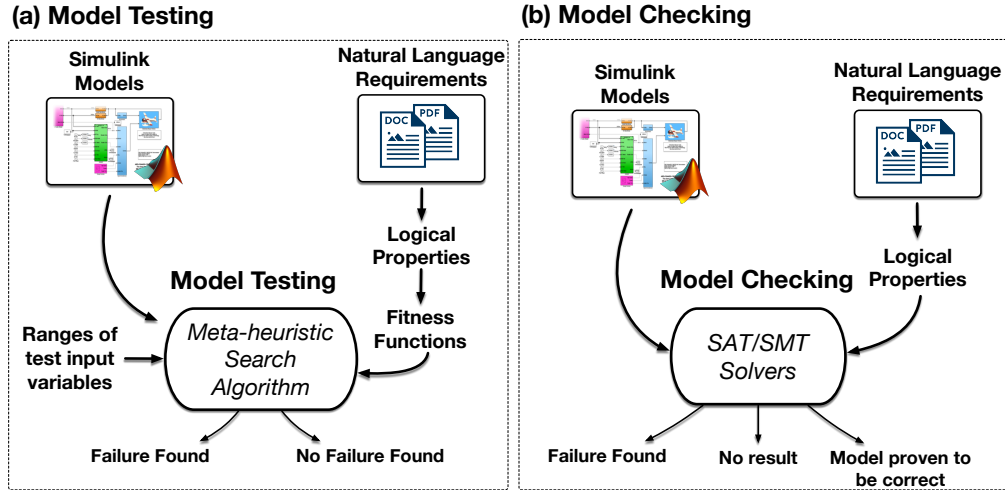


Figure 4.1: Simulink Model Verification: (a) Model testing and (b) Model checking.

2. We present the results of applying our model testing and model checking techniques to the Simulink benchmark. We evaluate the fault finding abilities of both techniques. This is a first attempt in the context of CPS, to systematically compare model checking and model testing – the two main alternatives for verifying Simulink models – on an industrial benchmark.
3. We provide some lessons learned where we outline the strengths and weaknesses of model testing and model checking in identifying faults in Simulink models. Since both approaches provide complementary benefits, we believe that integrating them in a comprehensive verification framework can result in an effective testing framework. We further propose some guidelines as to how both approaches can be best applied.

Organization. Section 4.1 presents our Simulink model benchmark, our CPS model categorization and our CPS requirements patterns. Section 4.2 summarizes the model checking technique that we apply in our study using QVtrace model checker. Section 4.3 describes the model testing approach. Section 4.4 presents the empirical results. Section 4.5 discusses some lessons learned. Section 4.6 concludes the chapter.

4.1 Simulink Benchmark

In this section, we present the CPS Simulink models and the CPS requirements in our Simulink benchmark. Table 6.4 shows a summary of the models in the benchmark. First, we present two example models from the benchmark in more details. Then, we categorize the benchmark models based on their functions. Finally, we describe the logic language to formalize our CPS requirements and present a number of recurring logic-based patterns in the requirements formalizations.

Example Models

We highlight two example Simulink models from the benchmark: *Two-Tanks* [GFH16] and *Autopilot*. These two models represent two complete systems instead of components of a system. The two-tanks system contains two separate tanks holding liquid and connected via a pipe. The flow of incoming liquid into the first tank is controlled using a pump. The flow of liquid from the first tank to the second is controlled using a valve, and the flow of outgoing liquid from the second tank is controlled

Table 4.1: Important characteristics of our benchmark Simulink models (from left to right): (1) model name, (2) model description, (3) model type, and (4) number of atomic blocks in the model.

Model Name	Model Description	Model Type	#Atomic Blocks
Autopilot	Discussed in Section 4.1	Feedback-loop, continuous controller, plant model, non-linear, non-algebraic, matrix operations	1549
Neural Network	A two-input single-output predictor neural network model with two hidden layers arranged in a feed-forward neural network architecture.	Open-loop, machine learning	704
Tustin	A numeric model that computes integral over time.	Open-loop, non-linear (saturation and switches)	57
Regulators	A PID controller without the plant model.	Open-loop, continuous controller, non-linear (saturation, switches)	308
Nonlinear Guidance	A non-linear guidance algorithm that guides an Unmanned Aerial Vehicles (UAV) to follow a moving target respecting a specific safety distance.	Open-loop, non-linear (polynomial, switches)	373
System Wide Integrity Monitor (SWIM)	A numerical algorithm that computes warning to an operator when the airspeed is approaching a boundary where an evasive fly up manoeuvre cannot be achieved.	Open-loop, non-linear (sqrt, switches)	164
Effector Blender	A control allocation method, which enables the calculation of the optimal effector configuration for a vehicle.	Open-loop, non-linear (polynomial), matrix operations, non-algebraic (exponential functions)	95
Two Tanks	Discussed in Section 4.1	Feedback-loop, state machine, non-linear (switches)	498
Finite State Machine (FSM)	A finite state machine executing in real-time. Its main function is to put the control of aircraft in the autopilot mode if a hazardous situation is identified in the pilot cabin (e.g., the pilot not being in charge of guiding the airplane)	Open-loop, state machine, non-linear (switches)	303
Euler	An open-loop mathematical model that generates three-dimensional rotation matrices along the z-y- and x-axes of an Inertial frame in a Euclidean space.	Open-loop, non-algebraic (trigonometry), non-linear (polynomial), matrix operations	834
Triplex	A monitoring system that receives three different sensor readings from three redundant sensors used in a safety critical system. It determines, based on the values and differences of three values received at each time step, which sensor readings are trusted and what values should be sent to the safety critical system.	Open-loop, state machine, non-linear (switches)	481

using two different valves: one that lets liquid out in normal situations, and the other that is opened only in emergency conditions to avoid liquid overflow. The model of the two-tanks system includes one controller model for each tank that monitors the liquid height using three different sensors located at different heights in each tank. Depending on the liquid heights, each controller chooses to open or close valves to control the incoming/outgoing liquid flows. The two-tanks model further includes a complete model of the environment (i.e., the tanks and their sensors and actuators).

The autopilot system is a full six degree of freedom simulation of a single-engined high-wing propeller-driven airplane with autopilot. A six degree of freedom simulation enables movement and rotation of a rigid body in three-dimensional space. The autopilot simulator model is able to control the plane body to change position as forward/backward (surge), up/down (heave) and left/right (sway) in three perpendicular axes, combined with changes in orientation through rotation in three

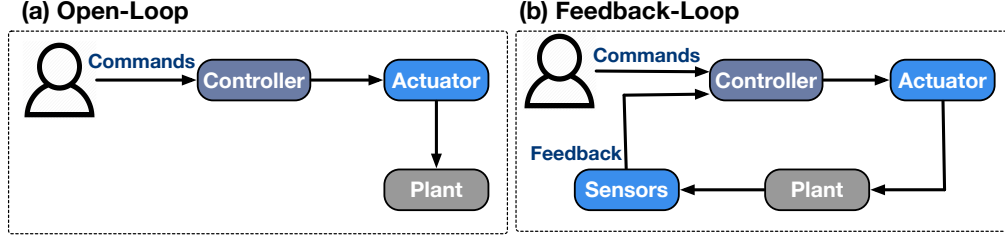


Figure 4.2: Generic structure of (a) open-loop and (b) feedback-loop CPS models

perpendicular axes, often termed yaw (normal axis), pitch (transverse axis) and roll (longitudinal axis). The autopilot model further captures a physical model of the airplane (i.e., a plant model) as well as environment aspects impacting airplane movements such as wind speed.

Both two-tanks and autopilot models use closed-loop controllers. However, the two-tanks controllers are modelled as discrete state machines, while the autopilot model consists of six continuous PID controllers [Nis04]. Some requirements of both models are described in Table 4.2

Table 4.2: Example requirements for the TwoTanks and Autopilot models.

Model	ID	Requirement	Signal Temporal Logic formula (STL) *
Two Tanks	R1	If the liquid height of the first tank is greater than or equal to the position of the top sensor of the first tank, then the top sensor should return an active (TRUE) state to the system.	$G_{[0,T]}(tank1height \geq tank1topSensor \Rightarrow tank1sensorHValue = 1)$
Two Tanks	R2	When the tank 2 MID sensor is TRUE, the tank 2 HIGH sensor is FALSE, and the emergency valve was previously OPEN, then the emergency valve and the production valve (outflow valves) shall be commanded to be OPEN.	$G_{[0,T]}((tank2sensorMValue = 1) \wedge (tank2sensorHValue = 0) \wedge (eValueStatePrev = 1) \Rightarrow (eValueState = 1) \wedge (pValueState = 1))$
Autopilot	R1	The controller output will reach and stabilize at the desired output value within %1 precision and within T seconds (steady-state requirement).	$F_{[0,T]}G_{[0,T]} out - desired \leq 0.01$
Autopilot	R2	Once the difference between the output and the desired value reaches less than %1, this difference shall not exceed 10% overshoot.	$G_{[0,T]}(out - desired \leq 0.01 \Rightarrow G_{[0,T]} out - desired \leq 0.1)$

* The variable T indicates the simulation time.

CPS Simulink Models Categorization

In the CPS domain, engineers use Simulink language to capture *dynamic systems* [Alu15]. We recall from Chapter 2, Section 2.2, the modeling language provided by Simulink and we explain how the systems design is constructed and simulated. Dynamic systems are usually used to model controller components as well as external components and the environment aspects that are to be controlled. The latter components are typically referred to as *plants*. Dynamic systems’ behaviors vary over time, and hence, their inputs and outputs are represented as signals (i.e., functions over time). We describe some common categories of dynamical system components that we have identified based on our industrial benchmark as well as Simulink models from other industrial sources [MNB17a]. We divide models into two large categories of *open-loop* and *feedback-loop* models:

(1) *Open-loop models* do not use measurements of the states or outputs of plants to make decisions [Alu15]. For example, an electronic cloth dryer controller that relies on time to change its states is an open loop model. The user sets a timer for the controller, and the dryer will automatically stop at the end of the specified time, even if the clothes are still wet. The design of such controllers

heavily relies on the assumption that the behavior of the plant is entirely predictable or determined. Figure 4.2(a) represents the generic structure of open-loop models.

(2) *Feedback-loop models* use measurements of the outputs or the states of plants to make decisions [Alu15]. This is the most common case in practical applications where engineers need to design system controllers that act on some controlled inputs depending on the current state of the plants. For example, an electronic cloth dryer controller that is able to stop when the clothes are sufficiently dried without requiring the user to set a timer, works by continuously monitoring the status of the clothes to choose when to stop the dryer. Such controllers are more flexible and are better able to handle unpredictable environment situations and disturbances. Figure 4.2(b) represents the generic structure of closed-loop models.

CPS models, whether being open-loop or feedback-loop, may consist of several components conforming to one or more of the following computation types:

State machines. State machines are used for modeling discrete and high-level controllers. They can be used to monitor system behaviour or to control the system either in an open loop or closed loop model. Three models in our benchmark use state machines: (1) Triplex is implemented using a state machine to monitor three different sensor readings from three redundant sensors and identifies errors in the sensor readings; (2) FSM uses an open-loop state machine controller to automatically put the control of aircraft in the autopilot mode if a hazardous situation is identified in the pilot cabin; (3) Two Tanks is implemented as a composition of two state machine controllers for each tank arranged in a feedback-loop architecture together with physical models of the two tanks. Each state machine controls pumps and valves of one tank. Since the pumps and valves can only have two states (i.e., on and off), they can be controlled using state machines with a few states. In general, state machines are useful to model systems that have a clearly identifiable set of states and transitions that, when fired, change the state of the system.

Continuous behaviors. Continuous mathematical models are used to describe both software controllers and physical plants. Continuous controllers, also known as *proportional-integral-derivative* PID-controllers [Nis04], are suitable when we need to control objects or processes whose states continuously vary over time. PID controllers are often used to control speed and movements of objects operating in varying environments with unpredictable disturbances. For example, the autopilot controller in Table 4.1 contains six PID controllers. Plant models, which are required for simulation of feedback-loop controllers, are typically described using continuous mathematical models. Continuous operations used in these two categories of models may have to be replaced with their discrete counterparts before the models can be translated into logic-based languages so that they can be analyzed by SMT-solvers. Though continuous controllers also need to be discretized for code generation purposes, this is not the case of plant models, and therefore, discretization of plant models is clearly an additional overhead.

Non-linear and non-algebraic behaviors. CPS Simulink models are typically highly numeric. They often exhibit non-linear behavior or may contain non-algebraic functions, making their analysis complicated. In particular, the following operations make Simulink models non-linear: saturation blocks, switches, polynomial and square-root functions, and the following operations are non-algebraic and are not typically supported by SMT-solvers: trigonometry functions, exponential functions and the logarithm. Finally, matrix operations are very commonly used in CPS Simulink models and are well-supported by Matlab. SMT-solvers, however, often do not directly support matrix operations, and hence, these operations have to be encoded and expanded as non-matrix formulas. Therefore, the size of the translations of Simulink models containing such operations into SMT-based languages become considerably larger than the size of the original models.

Machine learning models (ML). Machine learning models are often used at the perception layer of dynamical systems (e.g., for image processing) or are used to make predictions about certain

Table 4.3: Translation of Signal Temporal Logic [MN04] into quantitative fitness functions to be used in the model testing approach in Figure 4.1(a)

Translation to robustness metric [FP09] [MNGB19]	
$R_{(S,t)}(\top) = \epsilon$	$R_{(S,t)}(\perp) = -\epsilon$
$R_{(S,t)}(s^{\mathbb{B}}) = \begin{cases} \epsilon & \text{if } s^{\mathbb{B}} \\ -\epsilon & \text{if } \neg s^{\mathbb{B}} \end{cases}$	$R_{(S,t)}(\mu = 0) = - \mu(S_t) $
$R_{(S,t)}(\mu \neq 0) = \begin{cases} \mu(S_t) & \text{if } \mu(S_t) \neq 0 \\ -\epsilon & \text{else} \end{cases}$	$R_{(S,t)}(\mu \geq 0) = \mu(S_t)$
$R_{(S,t)}(\mu > 0) = \begin{cases} \mu(S_t) & \text{if } \mu(S_t) \neq 0 \\ -\epsilon & \text{else} \end{cases}$	$R_{(S,t)}(\mu \leq 0) = -\mu(S_t)$
$R_{(S,t)}(\mu < 0) = \begin{cases} -\mu(S_t) & \text{if } \mu(S_t) \neq 0 \\ -\epsilon & \text{else} \end{cases}$	
$R_{(S,t)}(\varphi_1 \vee \varphi_2) = \max(R_{(S,t)}(\varphi_1), R_{(S,t)}(\varphi_2))$	
$R_{(S,t)}(\varphi_1 \wedge \varphi_2) = \min(R_{(S,t)}(\varphi_1), R_{(S,t)}(\varphi_2))$	
$R_{(S,t)}(G_{[a,b]}\varphi) = \min\{R_{(S,t')}\{\varphi\}\}_{t' \in [t+a, t+b]}$	
$R_{(S,t)}(F_{[a,b]}\varphi) = \max\{R_{(S,t')}\{\varphi\}\}_{t' \in [t+a, t+b]}$	
$R_{(S,t)}(\varphi_1 U_{[a,b]}\varphi_2) = \max\{\min\{R_{(S,t')}\{\varphi_2\}, \min\{R_{(S,t'')}\{\varphi_1\}\}_{t'' \in [t, t']}\}_{t' \in [t+a, t+b]}$	

behaviors. Verification of models inferred by machine learning techniques (e.g., Neural Networks) is an open area for research, as exhaustive verification techniques have only been applied to relatively simple and small neural network models [KBD⁺17]. As shown in Table 4.1, we had one simple example machine learning component in our benchmark.

Table 4.1 describes the types of the models in the benchmark by specifying whether they are open-loop or feedback-loop and also by indicating what component or feature types are used in each model. We note that most models are specified as open-loop since they are in fact sub-systems of a larger system that may have a feedback-loop architecture. We note that the computation types described above are not meant to be exhaustive. Nevertheless, our categorisation provides more detailed information about the functional and behavioral variations in CPS models. Further, in the next subsection, we present the results of applying model checking and model testing approaches to our benchmark models, and the categorisation can further help determine how model checking and model testing approaches can deal with different computation types.

CPS Requirements and Patterns

As shown in Figures 4.1(a) and (b), both model checking and model testing require mathematical representations of requirements. Specifically, model checking expects requirements to be described in temporal or propositional logic, and model testing expects them to be captured as quantitative fitness functions. Requirements are properties the system must satisfy and usually constrain inputs and outputs behaviors.

We recall from Chapter 2, Section 2.1 the specification of CPS model requirements.

Before applying model testing or model checking, we first convert the textual requirements in the benchmark into their equivalent STL formulas. Model checking approaches typically receive a temporal logic property and a model as input. For model testing, however, we need to transform the logical properties into quantitative fitness functions (see Figure 4.1). To do so, we use a translation of STL into a robustness metric [FP09] which is summarized in Table 4.3. The translation function R is defined over a set $S = \{s_1, \dots, s_n\}$ of signals at time t . We assume that the signals in S are defined over the same time domain, i.e., for every $s_i \in S$, we have $s_i : [a, b] \rightarrow \mathcal{R}_i$ where $[a, b]$ is the

Table 4.4: Temporal patterns in STL translations of our benchmark requirements.

Name-ID	STL formula-tion	Explanation
Invariance - T1	$G\varphi$	The system should always exhibit the behaviour φ .
Steady State - T2	$F_{[0,d]}G\varphi$	The system within the time duration $[0, d]$ exhibits the behavior φ and continues exhibiting this behavior until the end.
Smoothness - T3	$G(\psi \Rightarrow G\varphi)$	Whenever the system exhibits ψ , it has to exhibit φ until the end.
Responsiveness - T4	$F_{[0,d]}\varphi$	The system shall exhibit φ within the time duration of $[0, d]$.
Fairness - T5	$GF_{[0,d]}\varphi$	At every time t , it should be possible for the system to exhibit the behaviour φ within the next time duration $[t, t + d]$.

common domain between signals in S . The choice of the *max* and *min* operators for defining the semantics of \exists and \forall is standard [LT88]: the minimum has the same behavior as \wedge and evaluates whether a predicate holds over the entire time interval. Dually, the max operator captures \vee .

For every STL property φ and every set $S = \{s_1, \dots, s_n\}$ of signals at time t , we have $R_{(S,t)}(\varphi) \geq 0$ if and only if φ holds over the set S at time t [FP09, MNGBI9]. That is, we can infer boolean satisfaction of STL formulas based on their fitness values computed by R . In Table 4.3, ϵ is an infinitesimal positive value that is used to ensure the above relation between boolean satisfiability and fitness values of real-valued constraints (i.e., μ **rel-op** 0) and literals (i.e., \top , \perp , and $s^{\mathbb{B}}$) in the STL grammar.

We translate the requirements in the benchmark Simulink models into STL. Some examples of STL formulas corresponding to the requirements in our benchmark are shown in Table 4.2. For example, the formula $F_{[0,T]}G_{[0,T]}(|out - desired| \leq 0.01)$ indicates that there is a time $t \in [0, T]$ such that for any time t' such that $t' \geq t$, the constraint $|out(t') - desired(t')| \leq 0.01$ holds. As an example, the R translation of this formula is given below:

$$\max\left\{\min\left\{0.01 - |out(t') - desired(t')|\right\}_{t' \in [t, t+T]}\right\}_{t \in [0, T]}$$

To provide more detailed information about the requirements in our benchmark, we present the recurring *temporal patterns* in the STL formulation of our benchmark requirements. Table 4.4 shows the temporal patterns we identified in our study. The invariance pattern, which simply states that a property should hold all the time, is the most recurring temporal pattern in our Simulink model benchmark. The other patterns in Table 4.4 capture common controller requirements, i.e., stability or steady-state, responsiveness, smoothness, and fairness. Note that in Table 4.4, the time interval for G operators are expected to be the same as the time domain of the signals to which the operators are applied. In Table 4.5, we show the list of the temporal patterns appeared in formalisations of the requirements of each model in our Simulink benchmark. As this table illustrates, the invariance pattern (**T1**) is used for some requirements of every model. The other temporal patterns (i.e., **T2**, **T3**, **T4**, and **T5**) only appear in requirements formalisations of models that include some continuous controllers (i.e. Autopilot and Regulator).

4.2 Our Model Checking Technique

SMT-based model checking has a long history of application in testing and verification of CPS models. Briefly, to check if a model M meets its requirement r , the requirement is first translated into a logical property φ . An SMT-solver is then used to prove satisfiability of $M \wedge \neg\varphi$. If $M \wedge \neg\varphi$ turns out to be SAT, then M does not satisfy φ . If $M \wedge \neg\varphi$ is UNSAT, it implies that M satisfies φ . In general, SMT-based model checkers are focused on checking safety properties (i.e., properties expressed using the G temporal operator). The liveness properties (i.e., properties that use the F

Table 4.5: Temporal patterns used in the requirements formalisations of each Simulink benchmark model.

Model	# Req	Patterns	Model	# Req	Patterns
Autopilot	11	T1, T2, T3, T4	Two Tanks	32	T1
Neural Network	3	T1	Tustin	5	T1
FSM	13	T1	Nonlinear Guidance	2	T1
Regulator	10	T1, T5	Euler	8	T1
SWIM	2	T1	Effector Blender	3	T1
Triplex	4	T1			

temporal operator) can be expanded assuming that they are specified over a finite time interval. For example, $F_{[0,d]}\varphi$ can be rewritten as $\bigvee_{t \in [0,d]} \varphi(t)$ assuming that $[0, d]$ is discrete time interval.

In our study, we use QVtrace as a representative SMT-based model checking tool for Simulink. In addition to the standard SMT-based model checking described above, QVtrace uses the k -induction technique [DHKR11] to enhance the set of formulas it can verify. QVtrace uses a logical predicate language referred to as QCT to capture requirements. QCT supports all the numerical and boolean operators described in STL grammar, but similar to most existing SMT-based model checkers, among the temporal operators, it only supports the temporal operator G , i.e., globally. Hence, among the temporal patterns in Table 4.4, QCT can specify **T1** and **T3** directly. Properties involving **T2**, **T4** and **T5** can be expressed in QCT after we expand them as discussed earlier. Specifically, as we will discuss in Section 4.4, the requirements that used temporal patterns **T2**, **T4** belong to Autopilot that could not be verified using QVtrace due to its complex features, and the requirement of the Regulator model that used the **T5** pattern was expressed in QCT as a large disjunctive formula (i.e., $G \bigvee_{t \in [0,d]} \varphi(t)$). We note that, in general, while being a subset of STL, QCT is sufficiently expressive for most problems we have seen in practice. In addition, QCT is carefully designed to be easy to read and understand by a typical engineer who may not have background in temporal logic. Finally, there is an efficient and straightforward translation from QCT into the input languages of SMT-solvers and theorem provers.

When the SMT-based formulation of $M \wedge \neg\varphi$ becomes so large that it cannot be handled by the underlying SMT-solvers, QVtrace relies on bounded model checking (BMC) [CBRZ01] mainly to identify inputs that falsify the model under test. BMC checks the behavior of the underlying model for a fixed number of steps k to see if a counter-example with length less than k can be found for the property of interest. As a result, BMC can only falsify properties up to a given depth k and is not able to prove the correctness of the model with respect to a property.

4.3 Our Model Testing Technique

We recall from Chapter 2, Section 2.3, the model testing technique that we apply in our study. In this chapter, we describe further our approach. it takes as input: (1) the model under test, (2) a fitness function guiding the search towards requirements violations, and (3) the value ranges of the model input variables. We discuss the fitness functions and the input search space below. We then present a well-known evolutionary search algorithm used in our work.

We use the robustness metric [FP09] as fitness functions in our work and use the translation in Table 4.3 to generate them from STL requirements formalizations. The robustness function $R(\varphi)$ is a value in $[-\infty, +\infty]$ such that $R(\varphi) \geq 0$ indicates that φ holds over model outputs (and hence the test satisfies φ), and $R(\varphi) < 0$ shows that φ is violated (and hence the test reveals a violation).

The robustness metric matches our notion of fitness as its value, when positive, shows how far a test input is from violating a requirement and when it is negative, its value shows how severe the failure revealed by a test is.

We use a simple evolutionary search algorithm, known as *hill climbing (HC)*, to generate test inputs (Algorithm 3). This algorithm has been previously applied to testing Simulink models [MNB⁺15b]. The algorithm receives the search input space characterization I and uses the fitness function f . It starts with randomly selected test input in the search space (CS selected in line 2). It then iteratively modifies the test input (line 4), compares its fitness value with the fitness value of the best found test input (line 5), and replaces the best test input when a better candidate is found (line 6). The search continues until an optimal test input (i.e., yielding a negative fitness value) is found or we run out of the search time budget. The test inputs in our work are vectors of boolean, enum or real variables. Hence, we implement the *Tweak* operator used in the HC algorithm by applying a *Gaussian Convolution* operator [Luk13] to the real variables and a *Bit-Flip* operator [Luk13] to the boolean and enum variables. The *Bit-Flip* operator randomly toggles a boolean or an enum value to take another value from its range. A *Gaussian Convolution* operator selects a value d from a zero-centered Gaussian distribution ($\mu = 0, \sigma^2$) and shift the variable to be mutated by the value of d . The value of σ^2 is in the order of 0.005 when we want to have an exploitative search operator (i.e., the one focused on locally searching a small area of the search space) and is selected to be higher (e.g., more than 0.1) when we are interested in more explorative search.

Algorithm 3 Hill Climbing Algorithm.

```

1:  $I$  : Input Space
2:  $CS \leftarrow$  initial candidate solution in  $I$ 
3: repeat
4:    $NS \leftarrow$  Tweak(Copy( $CS$ ))
5:   if  $f(NS) < f(CS)$  then
6:      $CS \leftarrow NS$ 
7: until  $CS$  is the ideal solution or we have run out of time
8: return  $CS$ 

```

4.4 Empirical Evaluation

In this section, we report the results of applying QVtrace tool (see Section 4.2) and our model testing technique (see Section 4.3) to our Simulink benchmark models described in Section 4.1. Specifically, we seek to answer the following research question: *How does model testing compare with (SMT-based) model checking in finding requirements violations in Simulink models?*

In the following, we explain the experimental setup we used for the evaluation. Then, we answer our research question based on the results.

Experiment Setup

As a prerequisite to apply both model testing and model checking to the benchmark Simulink models, we translated the textual requirements into STL. We performed this translation in collaboration with our industry partner. We had in total 92 requirements in our Simulink benchmark that we translated into STL. After that we used the translation in Table 4.3 to convert STL formulas into fitness functions to be used in our model testing approach. As discussed in Section 4.2, we further translated STL properties into QCT, the property language of QVtrace.

After converting textual requirements into fitness functions and formal properties, we applied model testing and model checking to the models to identify requirements failures. In the model testing technique, we used the HC algorithm discussed in Section 4.3. As discussed there, we used a *Gaussian Convolution* operator for the Tweak operation. In order for the HC search not to get stuck in local optima, we opt for a relatively large value of σ^2 for the Gaussian distribution from which the tweak values are chosen. Note that, in general, it is difficult to select a fixed value for σ^2 to tweak input variables of different models since these variables have different value ranges. Hence, for each real-valued input variable v , we set σ^2 to be 0.1 times the range width of v . We arrived at the value 0.1 through experimentation. If the tweaked values are out of variable ranges, we cap them at the max or min of the ranges when they are closer to the max or min, respectively. We set the main loop of HC (see Algorithm 3) to iterate for 150 times. We chose this number because, in our preliminary experiments, the HC search has always reached a plateau after 150 iterations in our experiments. Finally, in order to account for randomness in HC, for each model and for each requirement, we executed HC for 30 times.

To apply QVtrace, we first investigate whether it is applicable to the given model. If so, then QVtrace attempts, in parallel, to exhaustively verify the property of interest or to identify input values falsifying the property. The former is typically performed using k-induction and the latter is done using bounded model checking (BMC). QVtrace generates four kinds of outputs: (1) Green, when the property is exhaustively verified, (2) Red, when input values violating the property are found, (3) Blue, when the property is verified upto a bound k , and (4) Orange, when QVtrace fails to produce any conclusive results due to scalability or other issues. In this chapter, we present the results obtained based on the Z3 SMT solver [DMB08a] since it had better performance than other solvers.

Results

Table 4.6 reports the results of applying model testing and model checking to our Simulink model benchmark. Specifically, for model testing (MT), we report the number of requirements violations that we were able to find for each model. Recall that we executed HC 30 times for each requirement. Therefore, in Table 4.6, we report for each model and each violated requirement the number of fault revealing runs of MT. For example, out of 11 requirements in Autopilot, MT is able to identify five requirements violations. Three of these violations were revealed by 30 runs, one of them by four runs and the last by three runs. Since MT is black-box and analyzes simulation outputs, it is applicable to any Simulink model that can be executed. That is, it is applicable to all the benchmark models and requirements.

For model checking (MC), for each model, we report whether or not the model or all the requirements of a model could be analyzed by QVtrace (i.e., if the models and requirements could be translated into an internal model to be passed into SMT solvers). For the models and requirements that could be analyzed by QVtrace, we report in Table 4.6: (1) the number of requirements that can be checked exhaustively and proven to be correct, (2) the number of identified requirements violations, and (3) the number of requirements that were checked by bounded model checking (BMC) up to a specific bound k for which no violation was found.

For example, as shown in Table 4.6, QVtrace was not able to translate the Autopilot model. This is indicated in the table by showing that 0 out of the 11 requirements of Autopilot could be translated internally by QVtrace. However, QVtrace is able to handle Two Tanks and its 32 requirements. Among these, QVtrace proves 19 requirements to be correct, finds three requirements violations and is able to check ten requirements using BMC up to the following bounds, respectively: $k_1 \dots k_8 \approx 90$, $k_9 = 110$, and $k_{10} = 260$. Specifically, for these ten requirements of Two Tanks, BMC

Table 4.6: Comparing the abilities of model testing and model checking in finding requirements violations for Simulink models.

Model	# Reqs.	Model Testing (MT)		Model Checking (MC)			
		# Viola-tions	# Runs Revealing Vi-olations	# Trans-lated Reqs	# Proven Reqs	# Viola-tions	# Proven Reqs using BMC up to the Bound k
Autopilot	11	5	3(30/30), 1(4/30), 1(3/30)	0/11	-	-	-
Two Tanks	32	11	10(30/30), 1(29/30)	32/32	19	3 (11)*	10 ($k_1, \dots, k_8 \approx 90, k_9 = 110, k_{10} = 260$)
Neural Net-work	2	0	-	2/2	0	0	2 ($k = 0$)
Tustin	5	3	1(30/30), 1(29/30), 1(19/30)	5/5	2	2	1 ($k = 0$)
FSM	13	6	1(4/30), 1(6/30), 1(12/30), 1(9/30), 2(1/30)	13/13	7	6	0
Nonlinear Guidance	2	2	2(24/30)	2/2	0	2	0
Regulator	10	10	10(30/30)	10/10	0	9	1 ($k = 110$)
Euler	8	0	-	8/8	8	0	0
SWIM	2	0	-	2/2	2	0	0
Effector Blender	3	2	1(30/30), 1(1/30)	3/3	0	0	3 ($k = 0$)
Triplex	4	1	2(30/30)	4/4	3	1	0
Total:	92	40	-	81	41	23	17

* QVtrace is able to find three violations when it is applied to the original Two Tanks model. If we modify the Two Tanks model to move the tanks’ sensors closer together and to make the tanks smaller, QVtrace is able to find the eleven violations found by MT. This is because violations are revealed much earlier in the simulation outputs of the modified Two Tanks model than in the outputs of the original model.

is able to check the correctness of each requirement r_i up to the depth k_i (where $1 \leq i \leq 10$), but the underlying SMT-solver fails to produce results for any depth $k > k_i$ due to scalability issues. We note that, for Two Tanks, QVtrace is able to find all the 11 violations found by MT if the Two Tanks model is modified such that the tanks are smaller and the tanks’ sensors are closer together. This is because violations are revealed much earlier in the simulation outputs of the modified Two Tanks model than in the outputs of the original model. Finally, for some of the requirements of some models (i.e., Neural Network, Tustin and Effective Blender), BMC was not able to prove the requirements of interest for any bound k . In Table 4.6, we use $k = 0$ in the BMC column to indicate the cases where the SMT-solver failed to produce any results for $k = 1$ when the BMC mode of QVtrace is used.

Table 4.7 compares the time performance of running MT and MC. On average, across all the models, each run of MT took 5.8min. The maximum average execution time of an MT run (i.e., 150 iterations of the HC algorithm) was 18.5min (for Autopilot), and the minimum average execution time of MT was 3min (for Nonlinear Guidance). We note that the time required for running MT depends on the number of search iterations (which in our work is set to 150) as well as the time required to run a model simulation. The latter depends on the complexity of the model and the length of the simulation time duration. Every Simulink model in the benchmark already has a default simulation time duration that we used in our experiments.

Proving each of the 41 requirements in the benchmark, which could be exhaustively checked by MC, took only 0.63sec on average. The Two Tanks requirements required the longest average time to be proven (1.89sec), and the Euler requirements required the lowest average time to be proven (0.06sec). On average, it took MC 2.19sec to find 29 requirement violations in the benchmark. For

4. EVALUATING MODEL TESTING AND MODEL CHECKING FOR FINDING REQUIREMENTS VIOLATIONS IN SIMULINK MODELS

the 17 requirements where BMC had to be tried, we have listed the time it took for the BMC mode of QVtrace to report an “inconclusive” output when we try a bound k larger than the maximum bound values that BMC could handle and are shown in Table 4.6. We note that as shown in Table 4.7, there are variations in the time required by QVtrace to report “inconclusive”. In particular, in some cases, it takes several minutes or even hours to report the “inconclusive” message and in some cases, the message is reported after a few seconds. This has to do with the internal choices made in QVtrace, but in either case, the “inconclusive” message indicates that the underlying SMT-solver (i.e., Z3) is not able to report results either because the input to the SMT-solver is too large or because the solver cannot handle some features in its input.

Table 4.7: Comparing the time performance of model testing and model checking.

Model	avg. Time per MT run	avg. Time to prove reqs (MC)	avg. Time to violate reqs (MC)	BMC Time when QVtrace reports “inconclusive” for bound values k larger than the ones reporter in Table 4.6
Autopilot	18.5min	-	-	-
Two Tanks	5.1min	1.89s	1.09s	For the ten requirements of Two Tanks that have to be checked by BMC, QVtrace reports “inconclusive” after approximately 5min.
Neural Network	5.9min	-	-	QVtrace reports “inconclusive” for the two requirements of Neural Network after waiting for 1958.9s (32.6min) and 847.1s (14.1min), respectively.
Tustin	4.6min	0.19s	0.76s	QVtrace reports “inconclusive” for one requirement of Tustin after waiting for 1121s (18.7min) .
FSM	3.6min	0.59s	0.18s	-
Nonlinear Guidance	3min	-	0.12s	-
Regulator	3.6min	-	10.1s	QVtrace reports “inconclusive” for one requirement of Regulator after waiting for 1303.3s (21.7min).
Euler	4.5min	0.06s	-	-
SWIM	5.2min	0.18s	-	-
Effector Blender	4.4min	-	-	QVtrace reports “inconclusive” for two requirements of Effector Blender after waiting for 9475.4s (2.6h) and 4371.9s (1.2h), respectively. For the third requirement of Effector Blender, QVtrace reports “inconclusive” after only 37.8s.
Triplex	5.6min	0.88s	0.88s	-
Average	5.8min	0.63s	2.19s	-

We note that all the requirements violations were communicated to Company A who developed the benchmark and were confirmed as valid violation cases. The results show that all the violations discovered by MC were also discovered by MT, but there were violations that MT could discover that could not be identified by MC. Specifically, there were 17 violations that MT could find but not MC. Among these, five belonged to the Autopilot model that could not be handled by MC. The other 12 were among the 17 requirements that had to be checked by BMC, but BMC could not check the requirements beyond some bound k while the failures could be revealed by MT at a time step beyond k . Finally, we note that MC was able to exhaustively prove 41 requirements, whereas MT, being a testing framework, is focused on fault-finding only. In Section 4.5, we discuss the complementary nature of MT and MC and will draw a few lessons learned based on our results.

In summary, out of the 92 requirements in our benchmark, MT was able to identify 40 requirement violations and MC only found 23 of them, without detecting additional violations. Among the 40 violations found by MT, 32 were found by more than half of the runs. This shows, as we have seen before, that one should run MT as many times as possible. Among the 92 requirements, MC was able to prove correctness for 41 of them. Finally, MC and MT together were able to either prove or find violations for 81 of the 92 requirements.

4.5 Lessons Learned

We draw five lessons learned based on our experiment results and our experience of applying MT and MC [Nej20] to the Simulink benchmark. Our aim is to identify strengths and weaknesses of the two techniques when they are used to verify Simulink models, and provide recommendations as to how MC and MT can be combined together to increase effectiveness of Simulink verification.

Lesson1: *MC may fail to analyse some CPS Simulink models.* As confirmed by QRA, the most serious obstacle in adoption of model checking tools by the CPS industry is that such tools may not be readily applicable to some industrial Simulink models. In particular, the inapplicability issue is likely to happen for models capturing continuous and dynamical systems (e.g., Autopilot). Before one can apply a model checking tool, such models have to be decomposed into smaller pieces, their complex features have to be simplified and the black-box components may have to be replaced by concrete implementations. We note that Autopilot could be analyzed by QVtrace after removing the wind subsystem and discretising some computations (e.g., by replacing \int with sum or df/dt with $\Delta f/\Delta t$). However, such simplifications and modifications may not be always feasible because: (1) The simplifications may modify the model behavior and may compromise analysis of some system requirements. This undermines the precision of analysis performed by MC, and further, some system requirements that are related to the simplified or removed subsystems can no longer be checked by MC. (2) Such changes are expensive and require additional effort that may not be justified in some development environments.

Lesson2: *Bounded model checking may fail to reveal violations that can be, otherwise, easily identified by MT.* In our study, bounded model checking (BMC) has been successfully used for analysis of 17 requirements to which model checking could not be applied exhaustively. MT, however, was able to reveal violations for 12 of these 17 requirements. All these violations were obviously revealed at time steps greater than the selected bounds k in BMC. For example, for Two Tanks, MT was able to violate eight requirements that were proven to be correct by BMC up to a bound less than 270. But these violations could be revealed at around 500 and 1000 time steps of Two Tanks outputs.

Lesson3: *MC executes considerably faster than MT when it can prove or violate requirements. However, MC may quickly fail to scale when models grow in size and complexity.* MC executes considerably faster than MT when it can conclusively prove or violate a requirement and does not warrant the use of BMC. While it took MC less than a couple of seconds, on average, to prove properties or to find violations for the benchmark, the quickest run of MT took about 3min. While for small models, MC is quicker than MT, this trend unlikely holds for larger and more complex models. In particular, MC has the worst performance for Autopilot, Neural Network and Effector Blender that have complex features such as continuous dynamics, non-algebraic functions and machine learning components. Some of the limitations, however, are due to the underlying SMT-solvers.

Lesson4: *MT approaches, though effective at finding violations, need to be made more efficient on large models.* In this chapter, we used a relatively simple model testing approach implemented based on a Hill-Climbing algorithm guided by existing fitness functions proposed in the literature. MT approaches can be improved in several ways to increase their effectiveness and practical usability. In particular, MT is computationally expensive as it requires to run the underlying model a large number of times. Since different runs of MT are totally independent, an easy way to rectify this issue is to parallelize the MT runs, in particular, given that multicore computers are now a commodity. In addition, there are several strands of research that investigate different search heuristics or attempt to combine search algorithms with surrogate models, to reduce their computational time (e.g., [ANBS16, MNBB14]).

Lesson5: *More empirical research is required to better understand what search heuristics should be used for what types of models.* Engineers are provided with little information and guidelines as to how they should select search heuristics to obtain most effective results when they use MT. Each run of MT samples and executes a large number of test inputs. The generated data, however, apart from guiding the search, is not used to draw any information about the degree of complexity of the search problem at hand or to provide any feedback to engineers as to whether they should keep running MT further or whether they should modify the underlying heuristics of their search algorithms. We believe further research is needed in this direction to make MT more usable and more effective in practice.

Combining MC and MT. Our experience shows that MT and MC are complementary. MC can effectively prove the correctness of requirements when it is able to handle the size and the complexity of the underlying models and properties, while MT is effective in finding requirements violations. Indeed, for our benchmark, MC and MT together are able to prove 41 requirements and find 40 violations, leaving only 11 requirements (i.e., 12%) inconclusive. Given that MC is quite fast in proving and violating requirements, we can start by applying MC first and then proceed with MT when models or requirements cannot be handled by MC or its underlying SMT-solvers.

4.6 Conclusions

In this chapter, we applied our industrial Simulink model benchmark to evaluate and compare capabilities of model checking and model testing techniques for finding requirements violations in Simulink models. Our results show that our model checking technique is effective and efficient in proving correctness of requirements on Simulink models that represent CPS components. However, as Simulink models become larger and more complex, in particular, when they involve complex non-algebraic or machine-learning components or exhibit continuous dynamic behaviour, it becomes more likely that model checking or bounded model checking fail to handle them or identify faults in them. On the other hand, while our model testing technique can scale to large and complex CPS models and identify some of their faults, it is still computationally expensive and does not provide any guidelines on what search heuristics should be used for what types of models. In the end, we believe combining the two techniques is the best way ahead. We also believe more studies comparing the performance of these techniques in different contexts can help researchers better identify limitations and strengths of these two main-stream automated verification techniques. While model checking techniques are proved effective for finding failures in CPS, applying such techniques to complex industrial systems may become challenging. Exhaustive checking, in particular, is generally undecidable when applied to the entire cyber-physical systems or hybrid systems. As a consequence, in practice, exhaustive verification is applied to subcomponent of the system. In compositional verification, it is necessary to identify the environment information in which the subcomponent is expected to operate. In the next chapter, we propose an automated approach that combines search-based testing, machine learning and model checking to infer assumptions under which the component under analysis satisfies a given requirement.

Chapter 5

Mining Assumptions for Software Components using Machine Learning

In this chapter, we propose EPIcuRus (assumPtIon geneRation approach for CPS), an automated approach for synthesizing environment assumptions. EPIcuRus is tailored for 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 an 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 its 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 the input features with the 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.

The contributions of this work are summarized in the following:

1. We present the EPIcuRus assumption generation approach and provide a concrete and detailed implementation of EPIcuRus.
2. We formulate the assumption generation problem for Simulink models.

3. 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.
4. We introduce IFBT, a novel test case generation technique, that aims at increasing the efficiency and effectiveness of EPIcuRus.
5. 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.

Structure. Section 5.1 describes the challenges, provides motivating examples and outlines the pre-requisites of the proposed approach. Section 5.2 presents the approach of EPIcuRus. Section 5.3 formalizes the assumption generation problem. Section 5.4 describes the approach implementation. Section 5.5 evaluates EPIcuRus. Section 5.6 discusses the threats to validity of the proposed approach and Section 5.7 concludes the chapter.

5.1 Motivation and Challenges

We motivate our approach using `Autopilot`. We introduced `Autopilot` in Chapter 3, Section 3.3 and we described it in Chapter 4, Section 4.1. Our case study system 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 [Cha17]. The Simulink model of `Autopilot` is expected to satisfy a number of requirements, one of which is given below:

$\phi_1 ::=$ *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 altitude 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, 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 for which `Autopilot` does not satisfy ϕ_1 is that `Autopilot` is expected to operate under the following environmental assumption [Adm09]:

$\alpha_1 ::=$ *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 . 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 chapter, 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 [Cha09, Sim20] 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 part of `EPIcuRus`. In this chapter, we assume that requirements are expressed in an extension of Metric Temporal Logic (MTL) [Koy90b], named Signal Temporal Logic (STL) [MN04], where propositions of MTL are used to constrain the value assumed by the signals over time. This is an expressive language that captures complex properties, such as invariance, stability, responsiveness, smoothness, and fairness.

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

In this chapter, we consider `QVtrace` to exhaustively check whether a model M satisfies the requirement ϕ under the assumption A , i.e., $\langle A \rangle M \langle \phi \rangle$. `QVtrace` takes as input a Simulink model and a requirement specified in QCT which is a logical 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 a set of SMT-based model checkers (specifically Z3 BMC [DMB08b]) to verify Simulink models.

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

5.2 Approach Overview

Fig. 5.1 shows an overview of `EPIcuRus` which is described by Algorithm 4. `EPIcuRus` iteratively performs the following three main steps: (1) Test generation where a set TS of test cases that exercise 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 [CGP03]) to indicate that M restricted by A satisfies ϕ . If our model checker can assert $\langle A \rangle M \langle \phi \rangle$, an assumption

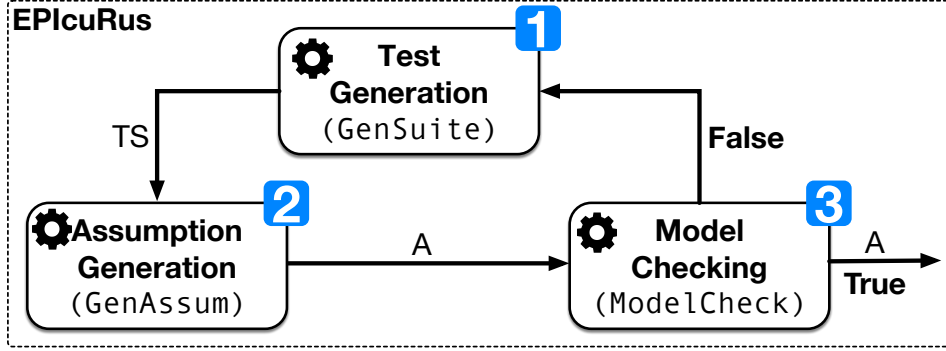


Figure 5.1: An overview on the EPIcuRus framework.

Algorithm 4 The EPIcuRus Approach.

Inputs. \mathcal{M} : The Simulink Model
 ϕ : Test Requirement of Interest
 opt : Options
 MAX_IT : Max Number of iterations
Outputs. A : The assumption

```

1: function A=EPICURUS( $\mathcal{M}$ ,  $\phi$ ,  $opt$ ,  $MAX\_IT$ )
2: Counter=0; TS= []; ▷ Variables Initialization
3: do
4: TS=GENSUITE( $\mathcal{M}$ ,  $\phi$ , TS,  $opt$ ) ▷ Test Suite Generation
5: A=GENASSUM(TS); ▷ Assumption Generation
6: Counter++; ▷ Increases the counter
7: while not MODELCHECK(A,  $\mathcal{M}$ ,  $\phi$ ) and Counter< $MAX\_IT$ 
8: return A;
9: end function

```

is found. Otherwise, we iterate the EPIcuRus loop. The approach stops when an assumption is found or a set time budget elapses.

Each model of our benchmark is a Simulink model that has a number of inputs and a number of outputs. We describe, in Chapter 2, Section 2.2, the modelling language provided by Simulink and we explain how the system design is constructed and simulated.

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 [HSX⁺19, AB11, CLM04] for test generation and we 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 chapter, we rely on classification trees to represent assumptions since our test cases are labelled by binary pass/fail values.

Fig. 5.2 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 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. Specifically, the path which traverses the nodes 1, 3, 7, 8 corresponds to the following conditions on the throttle and the pitch wheel ($0.44 < Throttle \wedge Throttle \leq 0.57 \wedge Pitchwheel \geq 5$), indicating that when the inputs `Throttle` and `Pitchwheel` of autopilot are constrained according to these conditions, the requirement ϕ_1 likely holds. Recall that the assumption α_1 explained earlier requires the `Throttle` value to be higher than a certain threshold to ensure requirement ϕ_1 . This matches the condition $0.44 < Throttle$ produced by the decision tree. The conditions $Pitchwheel \geq 5$ and $Throttle \leq 0.57$ may or may not be needed to ensure requirement ϕ_1 . In our approach, the path conditions that are typically produced by decision trees may involve additional constraints that may render assumptions too strong (restrictive) and hence has a lower coverage¹. As a result in this chapter, we are interested to produce the weakest assumptions (the ones with the largest coverage) that can guarantee our requirements.

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 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 [NGM⁺19]. 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 [DMB08b]) 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

¹ called *less informative* in our work [GMN⁺20]

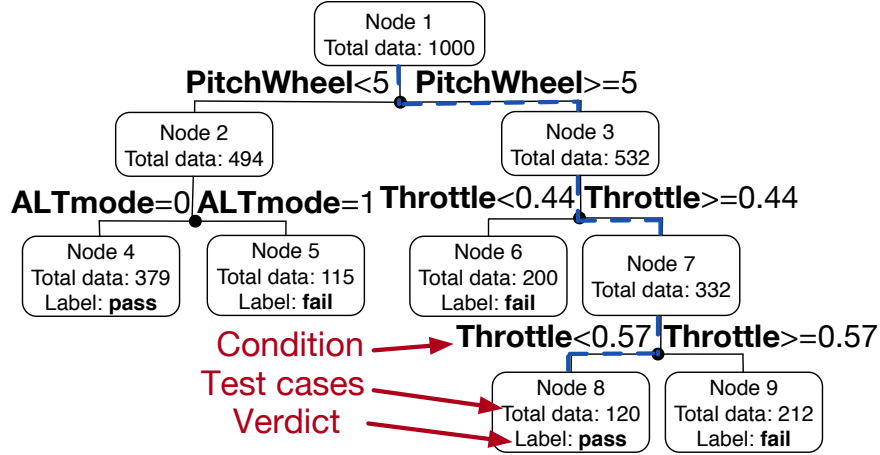


Figure 5.2: Example of classification tree constraining some of the inputs of the Autopilot running example.

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.

5.3 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 $C_1 \vee C_2 \vee \dots \vee C_n$ of one or more constraint $C = \{C_1, C_2, \dots, C_n\}$. Each constraint in C is a first-order formula in the following form:

$$\forall t \in [\tau_1, \tau'_1] : P_1(t) \wedge \forall t \in [\tau_2, \tau'_2] : P_2(t) \wedge \dots \wedge \forall t \in [\tau_n, \tau'_n] : P_n(t)$$

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 5.2 that $\mathbb{T} = [0, b]$ is the time domain used to simulate M . An example constraint for the `Autopilot` model discussed in Section 5.1 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 `ALTMMode` 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 `AC` model, which is described in Fig. 2.4a, 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 subdomains.

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

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

The input \bar{u} in Fig. 2.4a 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 $\bar{u} \in U$, we have $\bar{u} \models A$. Let ϕ be a requirement for M that we intend to verify. For every test input \bar{u} and its corresponding test output \bar{y} , we denote as $\llbracket \bar{u}, \bar{y}, \phi \rrbracket$ 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 [MNGB19, MNBP20, ALFS11b], we define the degree of violation or satisfaction as a function that returns 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 function $\llbracket \bar{u}, \bar{y}, \phi \rrbracket$ allows us to distinguish between different degrees of satisfaction and failure. When $\llbracket \bar{u}, \bar{y}, \phi \rrbracket$ is positive, the higher the value, the more requirement ϕ is satisfied, the lower the value the more requirement ϕ is close to be violated. Dually, when $\llbracket \bar{u}, \bar{y}, \phi \rrbracket$ is negative, a value close to 0 shows a less severe violation than a value close to -1 .

Definition 6 Let A be an assumption, let ϕ be a requirement for M . Let $U \subseteq \mathcal{U}$ be a valid input set of M restricted by assumption A . 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

$$v = \min_{\bar{u} \in U} \llbracket \bar{u}, \bar{y}, \phi \rrbracket$$

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

Definition 7 We say an assumption A is v -sound² for a model M and its requirement ϕ , if $\langle A \rangle M \langle \phi \rangle > v$.

As discussed earlier, we define the function such that a value v larger than or equal to 0 indicates that the requirement under analysis is satisfied. Hence, when an assumption A is sound, the model M restricted by A satisfies ϕ .

For a given model M , a requirement ϕ and a given value v , we may have several assumptions that are v -sound. We are typically interested in identifying the v -sound assumption that leads to the largest valid input set U , and hence is less constraining. Let A_1 and A_2 be two different v -sound assumptions for a model M and its requirement ϕ , and let U_1 and U_2 be the valid input sets of M restricted by the assumptions A_1 and A_2 . We say A_1 has a *larger coverage* than A_2 if it is weaker than A_2 , i.e., $A_1 \Rightarrow A_2$. In practical applications, there is an intrinsic tension between coverage and soundness. The larger the coverage of the assumptions, the less effective are exhaustive verification tools in proving their soundness. For example, QVtrace returns an inconclusive verdict for the requirement ϕ_1 when the assumption A_1 (see Section 5.1) is considered, i.e., the tool is neither able to prove that ϕ_1 nor to provide a counterexample showing that it does not hold. On the other hand, QVtrace can prove that the assumption A_2 is sound. Therefore, in industrial applications, users should find a practical *tradeoff* between coverage and soundness. We will evaluate this tradeoff for our microsatellite case study in Section 5.5.

In this chapter, provided with a model M , a requirement ϕ and a desired value v , our goal is to generate the weakest (that has the largest coverage) v -sound assumption. We note that our approach, while guaranteeing the generation of v -sound assumptions, does not guarantee that the generated assumptions have the largest coverage. Instead, we propose heuristics to maximize the chances of generating the assumptions that have the largest coverage and evaluate our heuristics empirically in Section 5.5.

²Called v -safe in our work [GMN⁺20].

5.4 Implementation

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

Test Generation

Algorithm 5 shows our approach to generate a test suite (1), i.e., a set of test inputs together with the associated 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 simulates the model on 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. We use existing techniques [MNGB19, ALFS11a] 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 $\text{FN}(\bar{u}, \bar{y}, \phi) \geq 0$, the output \bar{y} generated by the test input $\bar{u} = \{u_1, u_2 \dots u_m\}$ satisfies ϕ , and otherwise, \bar{y} and \bar{u} violate ϕ .

Specifying Test Cases. Algorithm 5 iteratively generates a new test input \bar{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.

To generate signals, we use the approach of Matlab [TFP⁺19, AWM⁺19] that encodes signals using some parameters. Specifically, each signal u_u in \bar{u} is captured by $\langle \text{int}_u, R_u, n_u \rangle$, 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 $c_{u,1}, c_{u,2}$,

Algorithm 5 Test Suite Generation.

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;                                     ▷ Test Suite Initialization
3:   for i=0; i<opt.TestSuiteSize; i++
4:      $\bar{u}_i$ =GENTEST( $\mathcal{M}$ , TSOld, opt);                 ▷ Test Case Generation
5:      $v_i$ =FN( $\bar{u}_i$ ,  $\mathcal{M}(\bar{u}_i)$ );                       ▷ Execute of the Test Case
6:     TSNew=TSNew $\cup$ { $\{\bar{u}_i, v_i\}$ };                 ▷ Add the Test to the Test Suite
7:   end for
8:   return TSNew
9: end function

```

\dots, c_{u,n_u} to assign to the control points for each input u_u . For example, the signals in Fig. 2.4a 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) [AB11, CKMT10, CLM04] and Uniform Random Testing (UR) [AIB12, AB11] 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.

Execution of the Test Cases. Our procedure uses the Simulink simulator (see Section 5.1) to generate output signals $\bar{y} = \mathcal{M}(\bar{u})$ associated with the generated test input \bar{u} . Using the fitness function FN, we obtain the verdict (pass/fail) value for each test input \bar{u} and output \bar{y} of the model under analysis. For example, the Simulink simulator generates the output in Fig. 2.4b from the input in Fig. 2.4a. The fitness value for this input and output computed based on the requirement ϕ_1 , described in Section 5.1, is ≈ 0.72 .

Assumption Generation

We use machine learning to automatically compute assumptions (2). Specifically, the function GENASSUM (see Algorithm 4) 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 5.3, 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 [Mol19a]. Recall from Section 5.4 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 [fit20] to build the trees. This function implements the standard CART algorithm [BFOS84]. EPIcuRus enables the users to select parameter values to configure the CART algorithm implemented by the `fitctree` function [fit20]. 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. 5.2 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. Setting the parameters of the DT learning is about avoiding both underfitting and overfitting, standard ML problems. In this chapter, we use default parameters provided by the Matlab library [fit20] since they yield reasonable results.

We follow a standard procedure to extract conditions on control points from a classification tree [WFHP16]. 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):

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

where $c_{x,y}$ denotes the y th 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 .

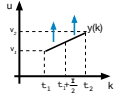
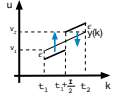
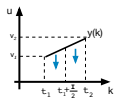
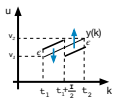
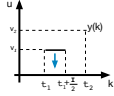
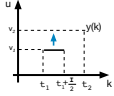
From conjunctions of predicates to a QVtrace assumption. The conjunctive predicates constrain control points. Before, we can use them with our model checker QVtrace, they have to be converted into constraints over signal variables as defined in Section 5.1. To do so, we use the rules provided in Table 5.1 to convert each predicate over control points, into constraints over signal variables. Specifically, the rules in Table 5.1 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 5.1 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 5.1, 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 5.1 and are presented in our online appendix [Epi20].

Intuitively, the conversion rules in Table 5.1 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 5.5), 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 5.1 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.

Table 5.1: Generating the predicates of the constraint.

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

* $t_1 = I \cdot j$ 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.

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 $A = C_1 \vee C_2 \vee \dots \vee C_n$ where each C_i is obtained by translating one conjunction of predicates using the rules in Table [5.1](#).

Model Checking

This step [\(3\)](#) aims at verifying whether a given assumption is v -sound. 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 [4.2](#), QVtrace generates four kinds of outputs. When it returns “No violation exists”, or “No violation exists for $0 \leq k \leq k_{max}$ ”, we conclude that the given assumption A is v -sound, i.e., the model under analysis when constrained by A satisfies the given formal requirement. Otherwise, we conclude that the assumption is not v -sound and has to be refined or modified.

IFBT — Important Features Boundary Test

Learning v -sound assumptions in our context requires a sufficiently large test suite, which is necessarily generated by simulating the 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 have the largest coverage for the ML procedure used to learn v -sound 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 -sound assumptions.

Algorithm 6 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:  $\langle$ Feat,Rng $\rangle$ =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 $\cup$ { $\langle \bar{u}_i, v_i \rangle$ };                                ▷ Add the Test to the Test Suite
10: endfor
11: return TSNew
12: end function

```

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 -sound assumptions.

Based on the above intuitive conjectures, IFBT generates a test suite by following the steps of Algorithm 6 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.

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 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

Table 5.2: 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

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.

5.5 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 5.4 and IFBT-UR and IFBT-ART discussed in Section 5.4) and determine which policy can help compute the most v -sound assumptions that have the largest coverage 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 -sound 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 [GMNB20].

Study Subjects. We consider eleven models and 92 requirements provided by Lockheed Martin [loc20, MNB17b, NGM⁺19, MNGB19]. The models have been described in details in Chapter 4, Section 4.1.

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**). Out of the 92 requirements, 27 could not be handled by QVtrace (violating (**Prerequisite-3**)) and 47 violated (**Prerequisite-4**). For 16 requirements, their Simulink models were not supported by QVtrace. For 11 requirements, QVtrace returned an inconclusive verdict due to scalability issues. Also, QVtrace could prove 46 requirements and refute one requirement for every input. We can conclude that for 27 requirements (30%), EPIcuRus cannot be applied due to the technical issues with MC. However, EPIcuRus is applicable to 65 (47+18) requirements (70%),

though in 47 (51%) of the cases MC is sufficient. We note that EPIcuRus is complementary to MC but, since it relies on MC as one of its components, it also inherits its limitations, i.e., only model components handled by MC can be targeted by EPIcuRus.

Thus, we retain only four models and 18 requirements. Table 5.2 describes the four models, the number of blocks and the inputs of each model and the number of requirements for each model. EPIcuRus is only needed when **Prerequisite-3** and **Prerequisite-4** hold since otherwise there is no need to generate assumptions. Prerequisite-3, however, is related to the scalability issues of applying QVtrace (or MC in general) to large/complex Simulink models.

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 5 and 6). We considered a maximum number of iterations MAX_IT=30 (see Algorithm 5.1) as this is a common practice to compare test case generation algorithms [EAD⁺19]. 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 -sound 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 -sound assumption, the computed v -sound 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 -sound assumption; and (ii) how large the coverage of 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 has a larger coverage 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 coverage 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 has a larger coverage than another assumption learned with a different test case generation policy. To check whether an assumption A_1 has a larger *coverage* 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 [BRP16, MBRP20]. 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 -sound assumptions due to a poor choice of initial inputs. Running all the experiments required approximately 33 days.³

RQ1 — Effectiveness and Efficiency

The scatter plot in Fig. 5.3 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 -sound 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 coverage index (`INF_INDEX`) associated to that policy. The higher this index, the larger the coverage of v -sound assumptions computed with a test case generation policy.

As shown in Fig. 5.3, 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 coverage index `INF_INDEX` is higher than those of the other policies.

Regarding the impact of using IFBT, Fig. 5.3 also shows that the difference between UR and IFBT-UR is small in terms of `V_SAFE` percentage, 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 av -sound 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 -sound assumption for $\approx 13\%$ more requirements.

The answer to **RQ1** is that, among the four test case generation policies we compared, IFBT-UR learns the most v -sound assumptions in less time. Further, the assumptions learned by IFBT-UR have a larger coverage than those learned by other test generation policies.

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 -sound 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 -sound 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.

³ We executed our experiments on the HPC facilities of the University of Luxembourg [\[VBCG14\]](#). The parallelization reduced the experiments time to approximately five days.

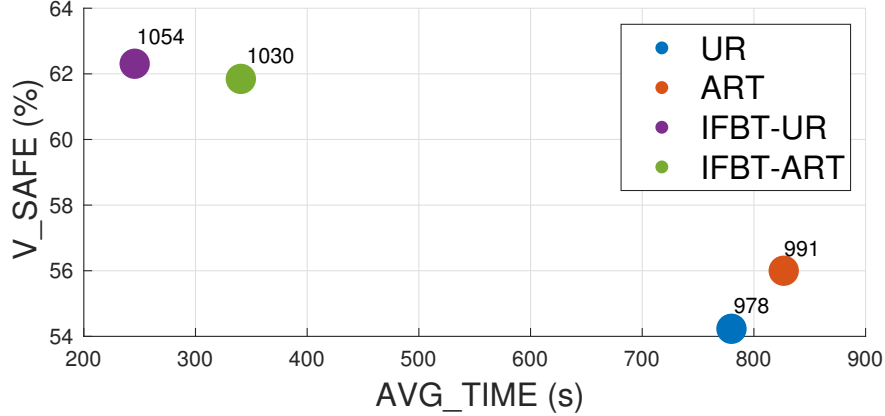


Figure 5.3: Comparison of the Test Case Generation Policies.

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.

5.6 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 [Mol19b] 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 [WFHP16], 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 [WFHP16]). Alternatively, we can use genetic programming [BNKF98] to learn more expressive assumptions.

- EPIcuRus generates assumptions using the rules in Table 5.1 that assume that consecutive control points are connected using a linear interpolation function. Our evaluation shows that this assumption provides a good compromise between simplicity and realism. The rules in Table 5.1 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 $\mathcal{O}(m \cdot n \cdot \log(n)) + \mathcal{O}(n \cdot (\log(n))^2)$. 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, are 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 [NGM⁺19, MNGB19], (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.

5.7 Conclusion

In this chapter, 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 works 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 areas with the largest coverage in the search space. Our evaluation shows that EPIcuRus is able to infer assumptions for all the requirements in our case 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.

In the next chapter, we extend our work to learn assumptions for complex CPS models involving signal and numeric variables, i.e., assumptions which include arithmetic expressions defined over multiple variables.

Chapter 6

Combining Genetic Programming and Model Checking to Generate Environment Assumptions

In this chapter, we improve the approach of EPIcuRus described in Chapter 5, Section 5.2. While EPIcuRus was effective in computing sound assumptions involving signal and numeric variables for industrial Simulink models, the structure of the assumptions generated by EPIcuRus was rather simple. Specifically, EPIcuRus could only learn conjunctions of conditions where each condition compares exactly one signal or numeric variable with a constant using a relational operator. This is because EPIcuRus uses decision tree classifiers that can only infer such simple conditions. In our experience, however, assumptions produced by EPIcuRus, while being sound, do not have the largest coverage that can be learned for many CPS Simulink models. We extend our previous work of EPIcuRus to learn an assumption that is not only sound (i.e., makes the component satisfy its requirements), but also has large coverage (i.e., is ideally the weakest assumption or among the weaker assumptions that make the component satisfy the requirement under analysis). We do so using genetic programming (GP) [KK92, PLMK08, BNKF98, GPM20, MAS05] where assumptions consist of conditions that relate multiple signals by both arithmetic and relational operators. EPIcuRus then applies model checking to the assumptions learned by GP to conclusively verify their soundness

This chapter highlights the following research contributions:

1. We formulate our assumption generation technique using GP. We provide a grammar for the assumptions that we aim to learn. For example, the actual assumption of our industrial case study—the attitude control component of the ESAIL maritime micro-satellite—is in the following form: $A_1 ::= \forall t \in [0, 1] : \alpha \cdot x(t) + \beta \cdot y(t) < c$, where x and y are signals defined over the time domain $[0, 1]$. But EPIcuRus, when relying on decision trees, is not able to learn any assumption in that form and instead learns assumptions in the following form: $A_2 ::= \forall t \in [0, 1] : \alpha' \cdot x(t) < c' \wedge \beta' \cdot y(t) < c''$. Assumptions in the latter form, even though sound, have lower coverage than the actual assumptions.
2. In the context of CPS, as assumptions have a larger coverage and become structurally complex,

establishing their soundness becomes more difficult as well. As discussed above, soundness can only be established via exhaustive verification (e.g., model checking). In our experience with industrial CPS Simulink models, model checkers fail to provide conclusive results by either proving or refuting a property when the assumption used to constrain the model inputs becomes structurally complex (e.g., when it involves arithmetic expressions over multiple variables). Therefore, the larger the coverage, the less effective exhaustive verification tools in proving their soundness, and vice-versa. Hence, if guaranteed soundness is a priority, engineers may have to put up with assumptions with lower coverage, and conversely, they can have assumptions with large coverage, whose soundness is not proven.

3. We evaluated EPIcuRus using two separate sets of models: First, we used the public-domain benchmark of Simulink models provided by Lockheed Martin [loc20] that we described in Chapter 3, Section 3.3 and in Chapter 5, Section 4.1. Second, we used the ESAIL model provided by LuxSpace [Lux19]. EPIcuRus successfully computed assumptions for 18 requirements of four benchmark models from Lockheed Martin [loc20] and one requirement of the attitude control component from LuxSpace. Note that, among all of our case study models, only these requirements needed to be augmented with environment assumptions to be verified by a model checker. Our evaluation targets two questions: if genetic programming (GP) can outperform decision trees (DT) and random search (RS) in generating sound assumptions that have large coverage (RQ1), and if the assumptions learned by EPIcuRus are useful in practice (RQ2). For RQ1, we considered all the 32 requirement and input profile combinations. Our results show that GP can learn a sound assumption for 31 out of 32 combinations of requirements and input profiles, while DT and RS can only learn assumptions for 26 and 22 combinations, respectively. The assumptions computed by GP have also a significantly (20% and 8%) larger coverage than those learned by DT and RS. For RQ2, we considered the attitude control component from LuxSpace since this is a representative and complex example of an industrial CPS component, and more importantly, in contrast to the public-domain benchmark, we could interact with the engineers that developed this component to evaluate how the assumptions computed by EPIcuRus compare with the assumptions they manually wrote.

In this chapter, we show that GP is able to generate assumptions that structurally conform to the grammar [Mon95], and in addition, maximize objectives that increase the likelihood of the soundness and coverage of the generated assumptions. EPIcuRus still applies model checking to the assumptions learned by GP to conclusively verify their soundness. The coverage, however, is achieved through GP and partly depends on the structural complexity of the learned assumptions. Any assumption that can be structurally generated by our grammar is in the search space of GP. Therefore, EPIcuRus with GP has more flexibility compared to the old version of EPIcuRus and can search through a wider range of structurally different assumption formulas to build more expressive assumptions that are likely to have a larger coverage as well.

Our results further show that, when EPIcuRus was configured to prioritize coverage, as opposed to proving the soundness of assumptions, learned assumptions were syntactically and semantically close to those written by engineers. Conversely, when learning assumptions whose soundness can be verified was prioritized, EPIcuRus was able to generate sound assumptions in around six hours. Though simpler than the actual assumption, they provided useful and practical insights to engineers. We note that none of the existing techniques for learning environment assumptions is able to handle our attitude control component case study or learn assumptions that are structurally as complex as those required for this component.

Structure. Section 6.1 introduces the motivating examples and challenges. Section 6.2 outlines the new approach of EPIcuRus and its pre-requisites. Section 6.3 formalizes the assumption gen-

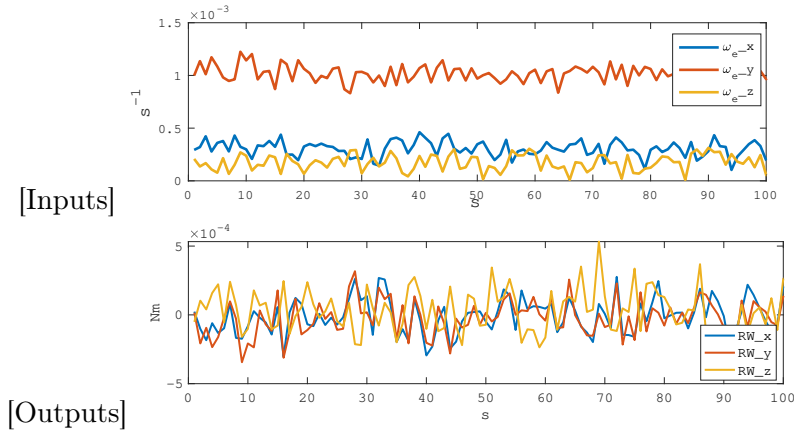


Figure 6.1: Example of Input/Output signals of the AC model.

eration problem and presents how EPIcuRus is implemented. Section 6.4 evaluates EPIcuRus, and Section 6.4 discusses the threats to validity. and Section 6.6 concludes the chapter.

6.1 Motivation and Challenges

As a case study system, we use the ESAIL maritime micro-satellite developed by LuxSpace [Lux19]. ESAIL is described in Chapter 3. The ESAIL Simulink model is a large, complex and compute-intensive model [MNBP20]. It contains 115 components (Simulink Subsystems [sub20]) and a large number (2817) of Simulink blocks of different types such as S-function blocks [sfu20] containing Matlab code, and some MEX functions [mex20] executing C/C++ programs containing the behavior of external third party software components. Due to the above characteristics, exhaustive verification of the ESAIL Simulink model (e.g., using model checking) is infeasible. For example, QVtrace cannot load the model of ESAIL since many components cannot be handled by the QVtrace model checker.

Even though the entire ESAIL Simulink model cannot be verified using exhaustive verification, it is still desirable to identify critical components of ESAIL that are amenable to model checking. For example, the Attitude Control (AC) component of the attitude determination and control system (ADCS) of ESAIL monitors the environment in which the satellite is deployed and sends commands to its actuators according to classical control laws used for the implementation of satellites [Wie98]. Specifically, it receives from the attitude determination system the estimated values of the speed, the attitude, the magnetic field and the sun measurements. It also receives commands from the guidance such as the target speed and attitude of the satellite. Then, it returns the commanded torque to the reaction wheel and the magnetic torquer. AC has ten inputs and four outputs that represent the commanded torque to be fed into the actuators. Note that some of the inputs and outputs are vectors containing several input signals, i.e., virtual vectors [vir20]. The inputs and outputs of AC are summarized in Table 6.1. For example, Fig. ?? shows the three input signals of ω_e , i.e., the estimated angular velocity of the satellite, over the time domain $[0, 100]$ s, for the AC component. Fig. ?? shows three output signals obtained by simulating the model with these inputs, and representing the torque command (RW) AC applied to the reaction wheels of the satellite over the time domain $[0, 100]$ s. AC contains 1142 blocks. It can be loaded in QVtrace after replacing the 19 S-function blocks, that cannot be processed by QVtrace, with a set of Simulink blocks supported by QVtrace. This activity is time-consuming and error prone. Every time an S-function is replaced by a set of Simulink blocks, to check for a discrepancy between the behaviors of the S-function and the newly added Simulink blocks, a set of inputs is generated and, for each input, the outputs

Table 6.1: Name of the Input, number of input signals in the virtual vector (NS), and description of the input.

	Name	NS	Description
Input	q_t	4	Target attitude of the satellite.
	q_e	4	Estimated attitude of the satellite.
	ω_t	3	Target speed of the satellite.
	ω_e	3	Estimated speed of the satellite.
	B	3	Measured magnetic field.
	Md	1	The mode of the satellite.
	Rwh	4	Angular momentum of the reaction wheel.
	Ecl	1	Indicates if the satellite is in eclipse.
	SF	1	Indicates if the sun sensor is illuminated.
	SM	3	Sun sensor measurements.
Output	MT	4	magnetic dipole applied to the magnetorquers.
	MTc	4	Current applied to the magnetorquers.
	RW	3	Torque applied to the reaction wheel.
	RWa	3	The reaction wheel's torque acceleration.

Table 6.2: Example requirements for the attitude control component.

ID	Requirement
ϕ_1	When the norm of the attitude error quaternion is less than 0.001, the torque commanded to the reaction wheel around the x-axis (with respect to its body frame) shall be within the range $[-0.001, 0.001] \text{N} \cdot \text{m}$.
ϕ_2	The magnetic moment of each magnetorquer shall be within the range $[-15, 15] \text{A} \cdot \text{m}^2$.
ϕ_3	The current applied to each magnetorquer shall be within the range $[-0.176, 0.176] \text{A}$.
ϕ_4	The acceleration of each of the reaction wheels shall be within the range $[-0.021, 0.021] \text{m/s}^2$.

produced by the S-function and the Simulink blocks is compared to check for dissimilarities. We did some testing to check for discrepancies between the behaviors of the S-function and the newly added Simulink blocks. We did not detect any error. After all the S-function blocks are removed, the model can be loaded in QVtrace, and we can have a formal proof of correctness (or lack thereof) for AC, which is an important component of ESAIL.

However, some requirements may fail to hold on AC when it is evaluated as an independent component, while the same requirements would hold on AC when it is evaluated within the larger model it is extracted from. This is because in the latter case the AC inputs are constrained by the values that can be generated within the larger model, which is not the case when AC is running independently. As a result, we need to verify whether the conditions under which AC works are acceptable given the input values that can be generated by its larger model. This is addressed by learning assumptions guaranteeing that AC satisfies its requirements.

For example, the Simulink model of the AC is expected to satisfy a number of requirements. Examples of these requirements are described in Table 6.2. The requirement ϕ_1 ensures that AC does not command any torque about the x-axis of the body frame to the reaction wheel, when the satellite is already at the desired attitude. Reaction wheels are used to control the attitude, i.e., the

orientation of the satellite, and ensure high pointing accuracy. They generate the twist applied to the satellite around a specific axis by acting on the acceleration of the reaction wheels. To determine whether, or not, AC satisfies requirement ϕ_1 , we convert the requirement into a formal property and use QVtrace to verify ϕ_1 over AC. However, it turns out that the requirement ϕ_1 does not hold on AC. Further, using QVtrace, we cannot show that AC satisfies $\neg\phi_1$, indicating that not all of its behaviors violate the requirement of interest. Therefore, for some inputs, AC violates ϕ_1 , and for some, it satisfies ϕ_1 . Note that if the model satisfies either ϕ_1 or $\neg\phi_1$, there is no need for generating an input assumption.

One of the reasons that the AC does not satisfy ϕ_1 is that, to ensure that the torque applied to the reaction wheel remains within $-0.001 \text{ N} \cdot \text{m}$ and $0.001 \text{ N} \cdot \text{m}$ when the norm of the attitude error quaternion is less than 0.001, we need to constrain the inputs of AC by the following assumption A_1 , which we elicited, in collaboration with the ESAIL engineers:

$$A_1 ::= \forall t \in [0, 1] : \overbrace{(\exp(t) + 1 \geq 0)}^{P_1(t)} \wedge \overbrace{(\exp(t) - 1 \leq 0)}^{P_2(t)}$$

where:

$$\begin{aligned} \exp(t) = & \overbrace{+783.3 \cdot \omega_e_x(t)}^{T1} \overbrace{-332.6 \cdot \omega_e_y(t)}^{T2} \overbrace{+3.5 \cdot \omega_e_z(t)}^{T3} \\ & - 50.57 \cdot \omega_e_x(t) \cdot \omega_e_y(t) - 4751.8 \cdot \omega_e_x(t) \cdot \omega_e_z(t) \\ & + 3588.7 \cdot \omega_e_y(t) \cdot \omega_e_z(t) \\ & + 1000 \cdot \text{Rwh_y}(t) \cdot \omega_e_z(t) - 1000 \cdot \text{Rwh_z}(t) \cdot \omega_e_y(t) \\ & \underbrace{-54.8 \cdot \omega_e_y(t)^2}_{T9} \underbrace{+54.8 \cdot \omega_e_z(t)^2}_{T10} \end{aligned}$$

The elicitation of the actual assumption was performed by two of the authors in collaboration with the ESAIL engineers. This was done by analyzing the design document of the satellite. The design document contains the decisions engineers made during the satellite design and specification. The elicitation of the assumption was performed by analyzing the specifications and identifying under which assumption the requirement is satisfied. This was done by computing the transfer function of the system, that describes the input-output relations, and by analytically identifying the most covering assumption that ensures the satisfaction of the requirement. Assumption A_1 constrains the values of the following variables within the time interval of $[0, 1]$ s: the estimated speed of the satellite (ω_e) and the angular momentum of the reaction wheel (Rwh) over the x-axis (ω_e_x and Rwh_x), the y-axis, (ω_e_y and Rwh_y) and the z-axis (ω_e_z and Rwh_z) of the body frame. This is done by forcing the value of exp to be between -1 and 1 . Predicates $P_1(t)$ and $P_2(t)$ are composed of the ten terms T1, T2, ..., T10 of exp and the constants 1 and -1 . For example, the term T3 of $P_1(t)$ is $+3.5 \cdot \omega_e_z(t)$.

Assumption A_1 is complex, and cannot be learned by our earlier work EPIcuRus [GMN⁺20] since it is a complex function that combines three input signals of ω_e and Rwh with arithmetic operators.

Requirement ϕ_2 in Table 6.2 constrains the magnetic moment commanded to the magnetorquers to be in the range $[-15, 15] \text{ A} \cdot \text{m}^2$. Requirement ϕ_3 constrains the current applied to the magnetorquers to be in the range $[-0.176, 0.176] \text{ A}$. Finally, requirement ϕ_4 constrains the torque acceleration applied to each of the reaction wheels to be in the range $[-0.021, 0.021] \text{ m/s}^2$. Those requirements were provided by the manufacturers of the reaction wheel and magnetorquer (see for example [Acq18]). According to the design documents, we expected these requirements to be satisfied for all possible input signals, i.e., without the need of adding any assumption.

Objective. Without accounting for assumption A_1 , we may falsely conclude that the AC model is faulty as it does not satisfy ϕ_1 . However, after restricting the inputs of AC with an appropriate

assumption, we can show that it satisfies ϕ_1 . Hence, there is no fault in the internal algorithm of AC.

In this chapter, we extend EPIcuRus to provide an automated approach to infer *complex* environment assumptions for system components such that they, after being constrained by the assumptions, satisfy their requirements. Our extension is applicable under three pre-requisites of EPIcuRus that we recall from Chapter 5, Section 5.1: **Prerequisite-1.** *The component M to be analyzed is specified in the Simulink language, **Prerequisite-2.** *The requirement ϕ the component has to satisfy is specified in a logical language* and **Prerequisite-3.** *The satisfaction of the requirements of interest over the component under analysis can be verified using a model checker.**

6.2 Approach Overview

In this section, we recall from Chapter 2, Section 2.2 the modelling language provided by Simulink and we explain how the system design is constructed and simulated. An *assumption* \mathbf{A} for a Simulink model \mathbf{M} is represented as a disjunction ($\mathbf{C}_1 \vee \mathbf{C}_2 \vee \dots \vee \mathbf{C}_n$) of one or more constraints in $\mathcal{C} = \{\mathbf{C}_1, \mathbf{C}_2, \dots, \mathbf{C}_n\}$. Each constraint in \mathcal{C} is a first-order formula in the following form:

$$\forall t \in [\tau_1, \tau'_1] : P_1(t) \wedge \forall t \in [\tau_2, \tau'_2] : P_2(t) \wedge \dots \wedge \forall t \in [\tau_n, \tau'_n] : P_n(t)$$

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. An example constraint for the AC model is defined as follows:

$$\mathbf{C}_1 ::= \forall t \in [0, 1] : (1.0 \cdot \omega_e_x(t) + 1.0 \cdot \omega_e_y(t) - 0.0011) \leq 0$$

\mathbf{C}_1 constrains the sum of the values of two input signals $\omega_e_x(t)$ and $\omega_e_y(t)$ of the input ω_e of the AC model over the time domain $[0, 1]$ s. These signals represent, respectively, the angular speed of the satellite over the x and y axes of the body frame.

Let $\mathbf{A} = \mathbf{C}_1 \vee \mathbf{C}_2 \vee \dots \vee \mathbf{C}_n$ be an assumption for model \mathbf{M} , and let \bar{u} be a test input for \mathbf{M} . The input \bar{u} satisfies the assumption \mathbf{A} if $\bar{u} \models \mathbf{A}$. For example, consider the assumption \mathbf{A}_2 of AC. The input \bar{u} in Fig. 2.4 satisfies the assumption \mathbf{A}_2 since it satisfies the constraints \mathbf{C}_1 and \mathbf{C}_2 .

$$\mathbf{A}_2 = \mathbf{C}_1 \vee \mathbf{C}_2$$

where:

$$\mathbf{C}_1 ::= \forall t \in [0, 1] : (1.0 \cdot \omega_e_x(t) + 1.0 \cdot \omega_e_y(t) - 0.0011) \leq 0$$

$$\mathbf{C}_2 ::= \forall t \in [0, 1] : (1.0 \cdot \omega_e_x(t) + 1.0 \cdot \omega_e_y(t) + 1.0 \cdot \omega_e_z(t) - 0.0025) \leq 0$$

Let \mathbf{A} be an assumption, and let \mathcal{U} be the set of all possible test inputs of \mathbf{M} . We say $\mathbf{U} \subseteq \mathcal{U}$ is a *valid input set* of \mathbf{M} restricted by the assumption \mathbf{A} if for every input $\bar{u} \in \mathbf{U}$, we have $\bar{u} \models \mathbf{A}$. Let ϕ be a requirement for \mathbf{M} that we intend to verify. For every test input \bar{u} and its corresponding test output \bar{y} , we denote as $\llbracket \bar{u}, \bar{y}, \phi \rrbracket$ the Boolean verdict indicating whether ϕ is satisfied or violated when \mathbf{M} is executed for test input \bar{u} .¹

Definition 8 *Let \mathbf{A} be an assumption, let ϕ be a requirement for \mathbf{M} , and $\llbracket \bar{u}, \bar{y}, \phi \rrbracket$ be the function previously discussed. Let $\mathbf{U} \subseteq \mathcal{U}$ be a valid input set of \mathbf{M} restricted by assumption \mathbf{A} . We say the satisfaction of the requirement ϕ over model \mathbf{M} restricted by the assumption \mathbf{A} is v , i.e., $\langle \mathbf{A} \rangle \mathbf{M} \langle \phi \rangle = v$, if*

¹The Boolean verdict is computed using existing approaches that extract the degree of violation or satisfaction of a property ϕ using existing techniques from the literature [MNGB19, MNBP20, ALFS11b].

$$v = \min_{\bar{u} \in U} \llbracket \bar{u}, \bar{y}, \phi \rrbracket$$

where \bar{y} is the test output generated by the test input $\bar{u} \in U$. For computing the min, we assume that true > false.

Definition 9 We say an assumption A is sound² for a model M and its requirement ϕ , if $\langle A \rangle M \langle \phi \rangle = \text{true}$

For a given model M and a requirement ϕ , we may have several assumptions that are sound. We are typically interested in identifying the sound assumption that leads to the largest valid input set U , and hence is less constraining. Let A_1 and A_2 be two different sound assumptions for a model M and its requirement ϕ , and let U_1 and U_2 be the valid input sets of M restricted by the assumptions A_1 and A_2 . In our previous work [GMN⁺20], we defined A_1 to be more informative than A_2 if $A_2 \Rightarrow A_1$. However, this definition only allows to compare assumptions when there exists a logical implication between the two. To enable a wider comparison among assumptions in this work, we define the notion of *coverage*. We say that A_1 has a larger *coverage* than A_2 if $|U_1| > |U_2|$ where $|U_1|$ and $|U_2|$ are, respectively, the cardinalities of the valid input sets U_1 and U_2 associated with A_1 and A_2 (i.e., the number of inputs in the sets U_1 and U_2). Our definition is generic and the sets can contain infinitely many inputs. What is important is to have a metric space, i.e., a set together with a metric on the set. For example, a real interval $[0, 1]$ is a metric space. Given a real interval $[0, 1]$, the "size" (or "measure" or "length") of the interval is 1 (the metric is the difference between the upper and the lower bound of the interval). This definition can be generalized to hypervolumes of polytopes, by extending the metric from a single dimension space to a multi dimension space. Our notion of coverage is compliant with the notion of logical implication. Given two assumptions A_1 and A_2 , such that $A_1 \Rightarrow A_2$, the assumption A_2 has a larger coverage than A_1 since it has a larger input set. Therefore, a more informative assumption (according to logical implication) has a larger coverage, i.e., a larger valid input set. The notion of coverage, however, is not limited to logical implication, and we can use it to compare assumptions that are not logically related, which is necessary in our context. We define the size of valid test inputs for an assumption as the number of valid test inputs within the test input set according to a given metric.

Note that computing the size of valid test inputs for our assumptions which are first-order formulas is in general infeasible. As we will discuss in Section 6.4, we provide an approximative method to compare the size of valid test inputs to be able to compare a given pair of assumptions based on our proposed coverage measure.

In practical applications, there is an intrinsic tension between coverage and soundness. The more assumptions with large coverage GP learns, the higher the chance that MC fails to confirm their soundness. This is because by increasing the coverage of the assumptions, they may turn unsound. For example, suppose the actual assumption that we want to learn is $x < 2$ (i.e., a sound assumption with an optimal coverage). GP may learn $x < 1$, which is sound but has a suboptimal coverage. In an attempt to increase coverage, GP is likely to produce an unsound assumption, e.g., $x < 2.1$, because the meta-heuristic search may not get to the exact value of 2. Therefore, in industrial applications, users should find a practical *tradeoff* between coverage and soundness. We will evaluate this tradeoff for our satellite case study in Section 6.4.

In this chapter, provided with a model M , a requirement ϕ and a desired value v , our goal is to generate the sound assumption that provides the largest coverage. We note that our approach, while guaranteeing the generation of sound assumptions, does not guarantee that the generated

²Called *v*-safe in our work [GMN⁺20], where we define the degree of satisfaction v

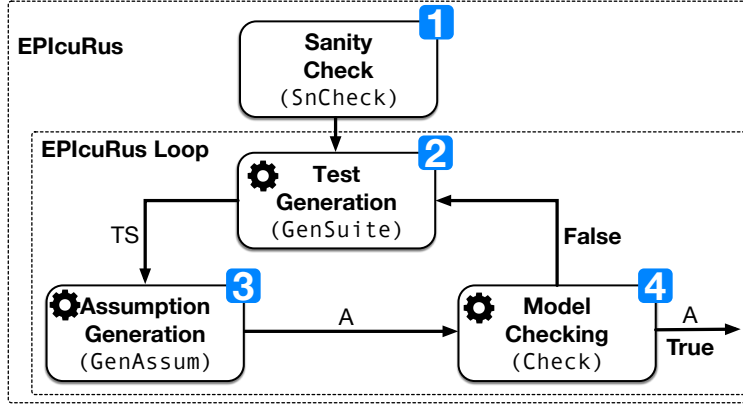


Figure 6.2: EPIcuRus framework overview.

assumptions have the largest coverage. Instead, we propose heuristics to maximize the chances of generating assumptions with the large coverage and evaluate our heuristics empirically in Section 6.4

Fig. 6.2 shows an overview of the extended approach of EPIcuRus, which takes as input a Simulink model M and a requirement ϕ , and computes an assumption ensuring that the model M satisfies the requirement ϕ when the assumption holds. The components of the approach are reported in boxes labeled with blue background numbers. The *sanity check* (1) verifies whether the requirement ϕ is satisfied (or violated) on M for all the inputs and therefore the assumption should not be computed. If the requirement is neither satisfied nor violated for all inputs, EPIcuRus iteratively performs the three steps of the EPIcuRus Loop (see Algorithm 7) discussed in the following:

2 *Test generation* (GENSUITE): returns a test suite TS of test cases that exercise M with respect to requirement ϕ . The goal is to generate a test suite TS that includes both passing (i.e., satisfying ϕ) and failing (i.e., violating ϕ) test cases;

3 *Assumption generation* (GENASSUM): uses the test suite TS to compute an assumption A such that M restricted by A is likely to satisfy ϕ ;

4 *Model checking* (CHECK): checks whether M restricted by A satisfies ϕ . We use the notation $\langle A \rangle M \langle \phi \rangle$ (borrowed from the compositional reasoning literature [CGP03]) to indicate that M restricted by A satisfies ϕ . If our model checker can assert $\langle A \rangle M \langle \phi \rangle$, a sound assumption is found.

There are two stopping criteria that can be selected for EPIcuRus. The first stopping criterion stops EPIcuRus whenever the model checker can assert $\langle A \rangle M \langle \phi \rangle$. The second stopping criterion constrains the timeout and allows EPIcuRus to refine the computed assumption over consecutive iterations.

A high-level description of each of these steps is presented in the following, a detailed description of the assumption generation procedure proposed in this work is provided in Section 6.3

Sanity Check

The sanity check (1) verifies whether the requirement ϕ is satisfied or violated for all the inputs. For this reason, we remove the **Prerequisite-4** considered in Chapter 5, Section 5.1. The sanity check uses a model checker to respectively verify whether $\langle \top \rangle M \langle \phi \rangle$ or $\langle \top \rangle M \langle \neg \phi \rangle$ is true. We use the symbol \top to indicate that no assumption is considered. If the requirement is satisfied for all inputs, no assumption is needed. If the requirement is violated for all inputs, then the model is faulty, and an assumption cannot be computed as there is no input that satisfies the requirement. The requirement *passes* the sanity check if some inputs satisfy ϕ while others violate it, i.e., the

Algorithm 7 EPICuRus Loop.

Inputs. M : the Simulink model
 ϕ : requirement of interest
 opt : options

Outputs. A : assumption

```

1: function  $A = \text{EPICURUSLOOP}(M, \phi, opt)$ 
2:    $TS = []$ ;  $A = \text{null}$ ; ▷ Variables Initialization
3:   do
4:      $TS = \text{GENSUITE}(M, \phi, TS, opt)$ ; ▷ Test Generation
5:      $A = \text{GENASSUM}(TS, opt)$ ; ▷ Assum.Gen.
6:      $A = \text{CHECK}(A, M, \phi)$ ; ▷ Model Checking
7:   while not  $opt.Stop\_Crt$ 
8:   return  $A$ ;
9: end function

```

requirement is neither satisfied nor violated for all inputs. In that case, the EPICuRus loop is executed to compute an assumption.

We use QVtrace to implement our sanity check. QVtrace exhaustively verifies whether a Simulink model M satisfies a requirement ϕ expressed in the QCT language, for all the inputs that satisfy the assumption A , i.e., $\langle A \rangle M \langle \phi \rangle$. QVtrace generates four different outputs (see Section 5.2).

- To check whether the model M satisfies the requirement ϕ for all inputs, we check $\langle \top \rangle M \langle \phi \rangle$. When it returns “No violation exists”, or “No violation exists for $0 \leq k \leq k_{max}$ ”, we conclude that the model under analysis satisfies the given formal requirement ϕ without the need to consider assumptions.
- To check whether the model M violates requirement ϕ for all inputs, we check $\langle \top \rangle M \langle \neg \phi \rangle$. If QVtrace returns “No violation exists”, or “No violation exists for $0 \leq k \leq k_{max}$ ”, we conclude that since $\neg \phi$ is satisfied, the model under analysis does not show any behavior that satisfies the requirement ϕ . Thus, the requirement ϕ is violated for any possible input, and the model is faulty.

In the two previous cases, EPICuRus provides the user with a value indicating that all the outputs of the model either satisfy or violate ϕ . Otherwise, the EPICuRus loop is executed to compute an assumption.

Test Generation

The goal of this step (2) 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 . Note that, while inputs that satisfy and violate the requirement of interest can also be extracted using model checkers, due to the large amount of data needed by machine learning (ML) to derive accurate assumptions, we rely on simulation-based testing for data generation. Further, it is usually faster to simulate models rather than to model check them. For example, performing a single simulation of AC and evaluating the satisfaction of ϕ_1 on the generated output takes 0.9s, while model checking AC against ϕ_1 takes approximately 21.06s. Hence, given a specific time budget, simulation-based testing leads to the generation of a larger amount of data compared to using model checking for data generation.

We use search-based testing techniques [HSX⁺19, AB11, CLM04] for test generation, which relies 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. We recall from Chapter 5, Section 5.4 the technique that we use to generate input signals. Specifically, we use the approach of Matlab [TFP⁺19, AWM⁺19] that encodes signals using some parameters. Specifically, each signal u_u in \bar{u} is captured by an input profile $\langle int_u, R_u, n_u \rangle$, 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. We assume that the number of control points is equal for all the input signals. Provided with the values for these three parameters, we generate a signal over time domain \mathbb{T} as follows:

1. We generate n_u control points, i.e., $c_{u,1}, c_{u,2}, \dots, c_{u,n_u}$, equally distributed over the time domain $\mathbb{T} = [0, b]$, i.e., positioned at a fixed time distance $I = \frac{b}{n_u-1}$. Let $c_{x,y}$ be a control point, x is the signal to which the control point refers, and y is the *position* of the control point. The control points $c_{u,1}, c_{u,2}, \dots, c_{u,n_u}$ respectively contain the values of the signal u at time instants $0, I, 2 \cdot I, \dots, (n_u - 2) \cdot I, (n_u - 1) \cdot I$;
2. We assign randomly generated values within the domain R_u to each control point $c_{u,1}, c_{u,2}, \dots, c_{u,n_u}$;
3. We use the interpolation function int_u to generate a signal that connects the control points. The interpolation functions provided by Matlab include, among others, linear, piece-wise constant and piece-wise 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 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 .

The verdict of the requirement of interest (ϕ) is then evaluated by (i) using the values assigned to the control points and the interpolation functions to generate input signals \bar{u} ; (ii) simulating the behavior of the model for the generated input signals and recording the output signals $\bar{y} = H(\bar{u}, M)$; (iii) evaluating the degree $\llbracket \bar{u}, \bar{y}, \phi \rrbracket$ of satisfaction of ϕ on the output signals; and (iv) labelling the test case with a verdict value (*pass* or *fail*) depending on whether $\llbracket \bar{u}, \bar{y}, \phi \rrbracket$ is greater (or equal) or lower than zero.

The test generation step returns a test suite **TS** containing a set of test cases, each of which containing the values assigned to the control points of each input signal and the verdict value. Depending on the algorithm used to learn the assumption, EPICuRus may or may not reinitialize the test suite **TS** at each iteration. In the latter case, the test cases that were generated in previous iterations remain in the new test suite **TS** that is also expanded with new test cases.

Assumption Generation

Given a requirement ϕ and a test suite **TS**, 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 . We use machine learning (ML) to derive an assumption based on test inputs labelled by binary verdict values. Specifically, the

assumption generation procedure infers an assumption by learning patterns from the test suite data (TS). This is done by (i) running the ML algorithm that extracts an assumption defined over the control points of the input signals of the model under analysis; and (ii) transforming the assumption defined over the values assigned to the control points into an assumption defined over the values of the input signals such that it can be checked by QVtrace.

Different ML techniques used in the assumption generation step lead to different versions of EPIcuRus. In this work, we consider Decision Trees (DT), Genetic Programming (GP), and Random Search (RS) as alternative assumption generation policies. DT is described in Chapter 5 [GMN⁺20], Section 5.4 while GP, and RS are described in Section 6.3 and are part of the contributions of this work.

Model Checking

This step checks whether the assumption A generated by the assumption generation step is sound. Note that the ML technique used in the assumption generation step, being a non-exhaustive learning algorithm, cannot ensure that assumption 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. QVtrace returns four possible results; “*No violation exists*”, “*No violation exists for $0 \leq k \leq k_{max}$* ”, “*Violations found*”, and “*Inconclusive*”. Specifically, when QVtrace returns “*No violation exists*” or “*No violation exists for $0 \leq k \leq k_{max}$* ”, we conclude that assumption A is sound, and hence ensures that M satisfies requirement ϕ . When QVtrace returns “*Violations found*”, we conclude that A is *theoretically* unsound, since M violates requirement ϕ under the assumption A . When QVtrace returns “*Inconclusive*”, the assumption can neither be proven sound nor theoretically unsound. In the remaining sections, we use the following terminology:

1. “*an assumption is sound*” denotes the cases in which the verdict “*No violation exists*” or “*No violation exists for $0 \leq k \leq k_{max}$* ” is returned by the model checker, i.e., it is proven that the assumption is sound.
2. “*an assumption is theoretically unsound*” denotes the cases in which the “*Violations found*” verdict is returned by the model checker, i.e., it is proven that the assumption is unsound.
3. “*an assumption is inconclusive*” denotes the cases in which “*Inconclusive*” verdicts are returned

6.3 Assumption Generation

In this section, we describe our solution for learning assumptions. For a given Simulink model M , we generate assumptions over individual control point variables of the signal inputs of M (Section 6.3). We then provide a procedure to lift the generated assumptions that are defined over control points to those defined over signal variables (Section 6.3). Our algorithm to generate assumptions uses Genetic Programming (GP) because we want to generate complex assumptions composed of arbitrary linear and non-linear arithmetic formulas. In addition, we introduce a baseline algorithm using Random Search (RS) for generating assumptions.

Learning Assumptions on Control Points with GP

Genetic programming (GP) is a technique for evolving programs from an initial randomly generated population in order to find fitter programs (i.e., those optimizing a desired fitness function). In our work, we use Strongly Typed Genetic Programming (STGP) [KK92, Mon95], a variation of GP

Algorithm 8 Genetic Programming (GP)

Inputs. TS: the test suite

opt: values of the parameters of GP (Table 6.3)

Outputs. A: assumption

```

1: function A=GENASSUM(TS,opt)
2:   t=0;
3:   P0=INITIALIZE(TS,opt);                                ▷ Initialize Population
4:   P0.Fit=EVALUATE(TS,P0);                              ▷ Assumptions Evaluation
5:   while t<opt.Gen_Size do
6:     Off=BREED(Pt,opt);                                    ▷ New Offspring
7:     Off.Fit=EVALUATE(TS,Off);                             ▷ Evaluation
8:     Pt+1=Off;                                             ▷ New Population
9:     t=t+1;
10:  endwhile
11:  A=BESTASSUM(P0, . . . ,Pt+1);                          ▷ Get Best Assum.
12:  return A;
13: end function

```

Table 6.3: Parameters of EPICuRus (EP) and its Genetic Programming algorithm (GP).

	Parameter	Description	Parameter	Description
EP	SBA	Search-based algorithm (GP, DT, RS).	ST	Simulink simulation time.
	TS_Size	The number of the generated test cases per iteration.	Stop_Crt	Stopping criteria: sound assumption found (MC) or timeout (Timeout).
	Timeout	EPICuRus timeout.	Nbr_Runs	Number of experiments to be executed.
GP	Max_Conj	Maximum number of conjunctions in an assumption.	Max_Disj	Maximum number of disjunctions in an assumption.
	Const_Min	Minimum constant value.	Const_Max	Maximum constant value.
	Max_Depth	Maximum depth of the syntax tree.	Init_Ratio	Percentage of the assumptions copied from the last population.
	Pop_Size	Number of individuals per population.	Gen_Size	Number of generations.
	Sel_Crt	The selection criterion.	T_Size	The number of individuals chosen for the tournament selection.
	Mut_Rate	Probability of applying the mutation operator.	Cross_Rate	Probability of applying the crossover operator.

designed to ensure that all the individuals within a population follow a set of syntactic rules specified by a grammar. The steps of our GP procedure are summarized in Algorithm 8. First (INITIALIZE), the algorithm creates an initial population containing a set of possible solutions (a.k.a. individuals). A fitness measure is used to assess how well each individual solves the problem (EVALUATE). In the evolutionary part, the algorithm iteratively generates new populations (BREED). It extracts a set of parents individuals and generates an offspring set by applying genetic operators to the parents. The algorithm then evaluates the individuals of the offspring (EVALUATE) and uses the offspring set as the new population P_{t+1} for the next iteration. The breeding and evaluation steps are repeated for a given number of generations (opt.Gen_Size). Then, the algorithm finds among all the individuals of the generated populations the individual with the highest fitness (BESTASSUM). The algorithm returns the individual with the highest fitness (A).

```

or-exp  :: = or-exp ∨ or-exp | and-exp
and-exp :: = and-exp ∧ and-exp | rel
rel     :: = exp (< | ≤ | > | ≥ | = ) 0
exp     :: = exp (+ | - | * | / ) exp | const | cp

```

Figure 6.3: Syntactic rules of the grammar that defines the assumptions on control points. The symbol `|` separates alternatives, `const` is a constant value, and `cp` is a variable that refers to a control point.

In the following, we describe how we use Algorithm 8 to generate assumptions over individual control point variables of the input signals of M .

Representation of the Individuals. Each individual represents an assumption over individual control point variables of the input signals of M . Specifically, assumptions are defined according to the syntactic rules provided by the grammar shown in Fig. 6.3. Furthermore, we constraint each arithmetic expression to contain only signal control points in the same position. For example, the assumption:

$$(c_{u_1,1} - c_{u_2,1} - 20 \leq 0) \vee ((c_{u_1,2} < 0) \wedge (c_{u_2,2} - 2.5 = 0))$$

is defined according to the grammar in Fig. 6.3. It constrains the values of the control points $c_{u_1,1}$, $c_{u_2,1}$, $c_{u_1,2}$, and $c_{u_2,2}$, and each arithmetic expression contains control points that refer to the same position. For example, $c_{u_1,1} + c_{u_2,1}$ contains the signal control points $c_{u_1,1}$, $c_{u_2,1}$ of the input signals u_1 and u_2 in position 1.

Initial Population. The initial population contains a set of `Pop_Size` individuals. The population size remains the same throughout the search. The method `INITIALIZE` generates the initial population. Recall from Section 6.2 (see Algorithm 7) that `GENASSUM` is called within `EPIcuRus` iteratively.

The first time that `INITIALIZE` is called it generates an initial set of randomly generated individuals. We use the `grow` method [PLMK08] to randomly generate each individual in the initial population. The `grow` method generates a tree with a maximum depth `Max_Depth`. It first creates the root node of the tree labeled with the Boolean operator \vee or \wedge or one among the relational operators $<$, \leq , $>$, \geq and $=$. Then, it iteratively generates the child nodes as follows. If the node is not a terminal, the algorithm considers the production rule of the grammar in Fig. 6.3 associated with the node type. One among the alternatives specified on the right side of the production rule is randomly selected and used to generate the child nodes. Then, the child nodes are considered. If the node is a terminal, depending on whether the node type is `const` or `cp`, a random constant value within the range `[Const_Min, Const_Max]` or signal control point is chosen with equal probability among the set of all the control points, respectively. To ensure that the generated tree does not exceed the maximum depth (`Max_Depth`) the algorithm constrains the type of the nodes that can be considered as the size of the tree increases. The algorithm also forces each arithmetic expression (`exp`) to only use control points in the same position. When all the nodes are considered the individual is returned.

In the subsequent iterations, however, `INITIALIZE` copies a subset of individuals from the last population generated by the previous execution of `GENASSUM`. The number of copied individuals is determined by the initial ratio (`Init_Ratio`) in its initial population and then randomly generates the remaining elements required to reach the size `Pop_Size`. This allows GP to reuse some of the individuals generated previously instead of starting from a fully random population each time it is executed.

Fitness Measure. Our fitness measure is used by the EVALUATE method to assess the soundness and coverage of the assumption individuals in the current populations (see Section 6.3). The fitness measure relies on the test cases contained in the test suite TS. We say a test case tc in TS satisfies an assumption A if tc is a satisfying value assignment for A. We compute the number of passing test cases in TS that satisfy A and denote it by TP. We also compute the number of failed test cases in TS that satisfy A and denote it by TN. We then compute the sound degree of an assumption A as follows:

$$sound = \frac{TP}{TP + TN}$$

The variable *sound* assumes a value between 0 and 1. The higher the value, the more passing test cases in TS satisfy the assumption. When *sound* = 1, all the test cases in TS that satisfy the assumption lead to a *pass* verdict. Since there is no failing test case in TS that satisfies the assumption, the assumption is likely to be *sound*.

To measure the coverage of an assumption, we compute the ratio of test cases in TS satisfying the assumption (TP+TN) over the total number of test cases in TS:

$$coverage = \frac{TP + TN}{TS}$$

The higher *coverage*, the weaker the assumption, and the more useful it is when informing engineers about when a requirement is satisfied.

Having computed *sound* and *coverage* for an assumption A, we compute the fitness function for A as follows:

$$FN = sound + \lfloor sound \rfloor \cdot coverage$$

where $\lfloor \cdot \rfloor$ is the Matlab floor operator [flo20]. This operator returns 0 for all the values of *sound* within the interval [0, 1) and returns 1 if *sound* is equal to 1. Function FN returns the *sound* value when *sound* is within the interval [0, 1). When *sound* is equal 1 it returns the value 1 + *coverage*. Intuitively, FN starts considering the coverage value only when an assumption is sound. This fitness function guides the search toward the detection of sound assumptions (which is our primary goal) that provide larger coverage.

We note that our fitness provides one way to combine *sound* and *coverage* values that fitted our needs. Developers may identify other ways to combine these two values to prioritize either *sound* assumptions or assumptions with large *coverage*.

Parents Selection. It uses the fitness values to select parent individuals for crossover and mutation operations (see SELECTPARENTS) that will be used to generate a new population. We implemented the following standard selection criteria of GP: *Roulette Wheel Selection (RWS)*, *Tournament Selection (TRS)* and *Rank Selection (RS)* [Lon11].

Genetic Operators. The genetic operators of GP act on the syntax tree of individuals. For example, the syntax tree of the following assumption is shown in Fig. 6.4

$$(c_{u_1,1} - c_{u_2,1} - 20 \leq 0) \vee ((c_{u_1,2} < 0) \wedge (c_{u_2,2} - 2.5 = 0))$$

Each node of the syntax tree represents a portion of the individual and is labeled (italic red label) with the identifier of the corresponding syntactic rule of the grammar of Fig. 6.3

The BREED method generates an offspring by

- either applying the *crossover* operator to generate two new individuals (with probability *Cross_Rate*) or randomly selecting an individual from P_t ; and

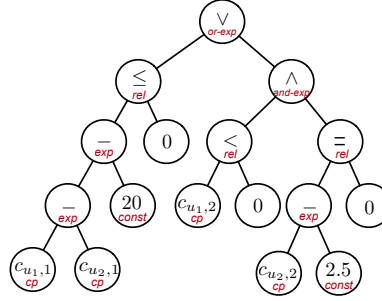


Figure 6.4: Syntax tree associated with an individual.

- applying the *mutation* operator (with probability `Mut_Rate`) to the individuals returned by the previous step.

The *crossover* and *mutation* genetic operators are summarized in the following.

We use one-point crossover [PL98] as *crossover* operator. One-point crossover (i) randomly selects two parent individuals; (ii) randomly selects one subtree in each parent; and (iii) swaps the selected subtrees resulting in two child individuals. To ensure that the child individuals are compliant with our representation, we force the following constraints to hold:

- The type of the root nodes of the subtrees is the same;
- The depth of the child individuals does not exceed `Max_Depth`;
- The number of conjunctions and disjunctions of the child individuals does not exceed `Max_Conj` and `Max_Disj`, respectively;
- When the type of the root nodes of the subtrees is `exp`, all the signal control points of the subtrees are in the same positions.

We use point mutation [PBL98] as a *mutation* operator. Point mutation mutates a child individual by randomly selecting one subtree and replacing it with a randomly generated tree. To create the randomly generated tree, we adopt the procedure used within the `INITIALIZE` method. Additionally, to ensure that the mutated child individual is compliant with our representation, our implementation ensures the constraints specified for the crossover operator are also satisfied here.

The full set of GP parameters is summarized in Table 6.3.

Random Search. It proceeds following the steps of Algorithm 8. However, at each iteration, a new set of individuals is randomly generated by adopting the same procedure used within the `INITIALIZE` method.

Control Points-Based to Signal-Based Assumptions

To use assumptions in QVtrace, it is necessary to translate assumptions that constrain control point values to assumptions that constrain signal values. To do so, we proceed as follows. Recall that control points $c_{u,1}, c_{u,2}, \dots, c_{u,n_u-1}, c_{u,n_u}$ are respectively positioned at time instants $0, I, 2 \cdot I, \dots, (n_u - 2) \cdot I, (n_u - 1) \cdot I$ and that any arithmetic expression contains only signal control points in the same position. Each expression `exp` that constrains the values of control points in position j is translated into an expression $\forall t \in [(j - 1) \cdot I, j \cdot I] : \text{exp}'$ where `exp'` is obtained by substituting each control point $c_{u_x,j}$ with the expression $u_x(t)$ modeling the input signal u_x at time t . Intuitively, this substitution specifies that the expression `exp` holds within the entire time interval $[(j - 1) \cdot I, j \cdot I]$.

For example, assuming that the control points 1, 2 and 3 are respectively positioned at time instant 0, 5 and 10, the assumption on the control points

$$(c_{u_1,1} - c_{u_2,1} - 20 \leq 0) \vee ((c_{u_1,2} < 0) \wedge (c_{u_2,2} - 2.5 = 0))$$

is translated into an assumption over the signal variables as follows:

$$(\forall t \in [0, 5) : u_1(t) - u_2(t) - 20 \leq 0) \vee ((\forall t \in [0, 5) : u_1(t) < 0) \wedge (\forall t \in [5, 10) : u_2(t) - 2.5 = 0))$$

Note that, our translation does not use the interpolation function of the input profile (see Section 6.2). Indeed, considering more complex interpolation functions may lead to assumptions that are less comprehensible and contain arithmetic functions that are more complex to interpret since they also relate signal values at different time instants. However, our approach can be extended to also consider the input profile for the translation from control-point-based assumptions to signal-based ones.

6.4 Evaluation

In this section, we evaluate our contributions by answering the following research questions:

- *RQ1 (Comparison of the search-based techniques).* How does GP compare with DT and RS in generating sound assumptions over signal variables with have a large coverage? (Section 6.4)

To answer this question, we compared the different search-based techniques of EPIcuRus and empirically assessed whether GP learns sound assumptions that have a larger coverage than the ones learned by DT and RS. We are not aware of any tool other than EPIcuRus for computing signal-based assumptions that we could use as a baseline of comparison. To answer this question we relied on a public-domain set of representative models of CPS components [MBG⁺20a] from Lockheed Martin [loc20]—a company working in the aerospace, defense, and security domains—and the model of our satellite case study (AC). Recall that EPIcuRus targets individual components, that can be analyzed using a model checker, and is generally not applicable to the entire industrial CPS models, such as the ADCS model (see Section 6.1).

- *RQ2 (Usefulness).* How useful are the assumptions learned by EPIcuRus?

To answer this question, we empirically assessed whether EPIcuRus can learn assumptions that specify valid inputs of a CPS component by comparing them with assumptions engineers would manually develop based on their domain knowledge and without any automated assistance. We answer this question by using our best search technique, according to RQ1 results, and the AC component of ADCS since 1. this is a representative example of an industrial CPS component (see Section 6.1), and 2. we could interact with the engineers that developed the AC component to evaluate how the assumptions computed by EPIcuRus compare with the assumptions they would manually write. To answer our research question, we considered the requirement ϕ_1 of AC (see Section 6.1) since EPIcuRus confirmed that the requirements ϕ_2 , ϕ_3 , and ϕ_4 , are satisfied for all possible input signals. Since there is, when dealing with complex components, a *tradeoff* among the coverage of the assumptions returned by EPIcuRus and the capability of QVtrace to confirm their soundness (see Section 6.2), engineers often have the choice to either learn assumptions with large coverage, whose soundness cannot be confirmed by a solver like QVtrace, or alternatively learn simpler assumptions, which have lower coverage but whose soundness can be verified exhaustively. Our goal is to investigate such tradeoff when analyzing industrial CPS components. Therefore, to answer RQ2, we are considering two sub-questions:

- *RQ2-1. How useful are EPIcuRUS assumptions when demonstrating their soundness is not a priority?*
- *RQ2-2. How useful are EPIcuRUS assumptions when learning assumptions whose soundness can be verified is prioritized?*

Implementation and Data Availability. We extended the original Matlab implementation of EPIcuRus [GMN⁺20]. We decided to implement the procedure presented in Section 6.3 by reusing existing tools. Among the many tools available in the literature (e.g., Weka [FHW16], GPLAB [SA03], GPTIPS [Sea15], Matlab GP toolbox [GPM20, MAS05]), we decided to rely on tools developed in Matlab. This facilitates the integration of our extension within EPIcuRus, and restricted our choice to GPLAB, GPTIPS, and the Matlab GP toolbox. Among these, we implemented our techniques on the top of GPLAB. We chose GPLAB since it allows the introduction of new genetic operators by adding new functions. We exploited this feature to implement the genetic operators of the procedure presented in Section 6.3. Our implementation and results are publicly available [Epi20], alongside the paper of the work presented in this chapter [GMN⁺55].

RQ1 — Comparison of the Search-Based Techniques

To compare GP, DT, and RS, we considered 12 models of CPS components and 94 requirements [MBG⁺20a]. These models include 11 models developed by Lockheed Martin [loc20] and the model of our satellite case study (AC). The models and requirements from Lockheed Martin were also recently used to compare model testing and model checking [NGM⁺19] and to evaluate our previous version of EPIcuRus [GMN⁺20]. Table 6.4 contains the description, number of blocks, inputs, and outputs of each CPS component model. It also contains the simulation time and the number of requirements considered for each model.

Out of the 94 requirements, 27 could not be handled by QVtrace (violating **Prerequisite-3**). For 16 requirements, the Simulink models of the CPS components were not supported by QVtrace. For 11 requirements, QVtrace returned an inconclusive verdict due to scalability issues. For 48 of the 67 requirements that can be handled by QVtrace, EPIcuRus did not pass the sanity check: QVtrace could prove 47 requirements and refuted one requirement. Therefore, to answer research question RQ1, we considered the 19 requirements of four models, that can be handled by QVtrace, pass the sanity check, and required the assumption generation procedure to be executed (column #Reqs of Table 6.4 within round brackets).

Methodology and Experimental Setup. To answer our research question, we configured the parameters of GP in Table 6.3 according to values in Table 6.5. We chose default values from the literature [PLMK08] for the population size (`Pop_size`), mutation rate (`Mut_rate`), crossover rate (`Cross_Rate`), and the max tree depth (`Max_Depth`) parameters. We set tournament selection (TRS) as selection criterion (`Sel_Crt`) since, when compared with other selection techniques, it leads to populations with higher fitness values [PLMK08]. We set the value of the tournament size (`T_Size`) according to the results of an empirical study on ML parameter tuning [AF13]. We set the maximum number of generations (`Gen_Size`), the number of conjunctions (`Max_Conj`) and disjunctions (`Max_Disj`), and the initial ratio (`Init_Ratio`) based on the results of a preliminary analysis we conducted, over the considered study subjects, where we determined the average number of generations needed to reach a plateau. We assigned to `Const_Min` and `Const_Max`, respectively, the lowest and highest values the input signals can assume in our study subjects. We set the number of tests in the test suite (`TS_Size`) to 300, which was the value used to evaluate falsification-based testing tools in the ARCH-COMP 2019 and 2020 competitions [EAD⁺19, EAB⁺20].

6. COMBINING GENETIC PROGRAMMING AND MODEL CHECKING TO GENERATE ENVIRONMENT ASSUMPTIONS

Table 6.4: Identifier (ID), name, description, number of blocks (#Bk), inputs (#In), outputs (#Out), simulation time (ST), number of requirements (#Reqs), and the number of requirements we used to answer research question RQ1, i.e., they pass the sanity check (#Reqs within round brackets) of each Simulink model of the components of our study subjects.

ID	Name	Description	#Bk	#In	#Out	ST(s)	#Reqs
TU	Tustin	A numeric model that computes integral over time.	57	5	10	10	5 (2)
EB	Effector Blender	A controller that computes the optimal effector configuration for a vehicle.	95	1	7	0	3 (0)
SW	Integrity Monitor	Monitors the airspeed and checks for hazardous situations.	164	7	5	10	2 (0)
FSM	Finite State Machine	Controls the autopilot mode in case of some environment hazard.	303	4	1	10	13 (2)
REG	Regulator	A typical PID controller.	308	12	5	10	10 (6)
NLG	Nonlinear Guidance	A guidance algorithm for an Unmanned Aerial Vehicles (UAV).	373	5	5	10	2 (0)
TX	Triplex	A redundancy management system.	481	5	4	10	4 (0)
TT	Two Tanks*	A controller regulating the incoming and outgoing flows of two tanks.	498	2	11	14	32 (8)
NN	Neural Network	A predictor neural network model with two hidden layers.	704	2	1	100	2 (0)
EU	Euler	Computes the rotation matrices for an inertial frame in a Euclidean space.	834	4	2	10	8 (0)
AP	Autopilot	A DeHavilland Beaver Airframe with Autopilot system.	1549	7	1	1000	11 (0)
AC	Attitude Control	Attitude control component of the ADCS of ESAIL.	438	10	4	1	2 (1)

* does not support multiple control points.

We configured RS by noticing that RS reuses part of the algorithm of GP. Therefore, for the parameters of RS, which are a subset of the parameters of GP, we assigned the same values considered for GP. Finally, we configured DT by considering the same values used by our earlier work, Gaaloul et al. [GMN⁺20], to evaluate EPIcuRus.

To answer RQ1, we performed the following experiment. We considered each of the 19 requirements under analysis and three different input profiles with respectively one (IP), two (IP'), and three (IP'') control points. We chose the number of control points of the input profiles based on the default input signals provided by the models of the CPS components. For the eight requirements of Two Tanks (TT), only the input profile IP was considered since this model only supports constant input signals.

Therefore, in total, we considered 32 requirement-profile combinations. For each combination, we ran EPIcuRus with GP, DT, and RS. We set a timeout of one hour, which is reasonable for this type of applications. As typically done in similar works (e.g., [EAD⁺19, MNBP20]), we repeated each run 100 times to account for the randomness of the test case generation procedure. Therefore,

Table 6.5: Values for the parameters of Table 6.3 used for RQ1.

	Parameter	Value	Parameter	Value
EP	SBA	DT/GP/RS	ST	see Table 6.4
	TS_Size	300	Stop_Crt	Timeout
	Timeout	1h	Nbr_Runs	100
GP	Max_Conj	3 or 4	Max_Disj	2
	Const_Min	-100	Const_Max	100
	Max_Depth	5	Init_Ratio	50%
	Pop_Size	500	Gen_Size	100
	Sel_Crt	TRS	T_Size	7
	Mut_Rate	0.1	Cross_Rate	0.9

* The values within the framed boxes , , and are respectively from [PLMK08], [AF13], and [EAD⁺19]. The value within the framed box is based on a preliminary analysis of the considered study subjects. The values within the framed boxes are selected based on our domain knowledge on the considered study subjects.

in total, we executed 9600 runs³: 3200 runs ($32 \cdot 100$) for each of GP, DT, and RS. For each run, we recorded whether a sound assumption was returned. Furthermore, we computed the coverage value associated with the assumption. To compute a coverage value (COV_V), we need to compute the size of the valid input set for each assumption (see $|U|$ in Definition 8). We do so empirically. Specifically, we generated 100 different value assignments for *each* control point. These assignments are uniformly distributed within the value range of the control point. Then, we create the input set that we use to evaluate all the generated assumptions. When the assumption constrains more than one control point, the input set contains all the possible combinations of the value assignments of the control points. For example, to evaluate an assumption that constrains two control points, we create an input set with 10000 assignments ($100 \cdot 100$). Each input in the set is a combination of two assignments, each selected from the 100 assignments of each control point. We then compute the percentage of the number of valid inputs in the set (i.e., the inputs that satisfy the assumption). The higher this number, the larger the coverage provided by the assumption.

Results. The results of our comparison are reported in Figure 6.5. Each box plot reports the coverage value (COV_V) of the assumptions computed by GP, DT, and RS, and is labelled with the percentage of runs, across the 3200 runs, in which the technique was able to compute a sound assumption (the value reported below the box plot). The average coverage values of the assumptions computed by GP, DT, and RS across their different runs are, respectively and approximately, 50%, 30% and 42%. Though there are variations across the different combinations of requirements and input profiles, GP can compute assumptions with a coverage value, that is, on average, 20% and 8% higher than that of the assumptions computed by DT and RS, respectively.

Across all the runs, GP was able, on average, to compute a sound assumption in 47.9% of the cases (1533 out of 3200), while DT, and RS were able to compute a sound assumption in respectively 46.7% (1495 out of 3200) and 48.9% (1565 out of 3200) of the cases. Therefore, GP is, on average, slightly more effective (1.2%) than DT, and slightly less effective than RS (1%) in computing sound assumptions. For each requirement-profile combination, Table 6.6 reports the number of runs, among the 100 runs executed for the requirement-profile combination, in which

³ We executed our experiments on the HPC facilities of the University of Luxembourg [VBCG14].

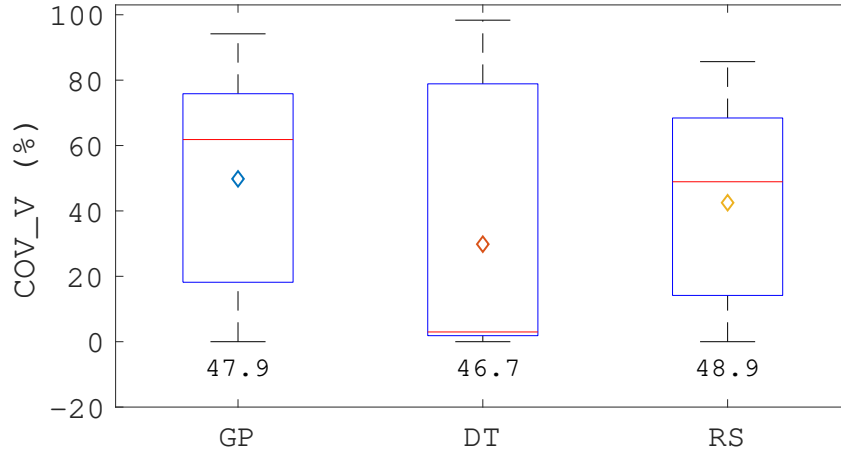


Figure 6.5: Comparing GP, DT, and RS. The box plots show the coverage value of GP, DT, and RS (labels on the bottom of the figure). Diamonds depict the average. The value below the box plot is the percentage of runs, across all the runs, in which the technique was able to compute a sound assumption.

GP, DT, and RS were able to compute a sound assumption. When GP was less effective than DT and RS, in many runs, across all the different iterations, GP was able to learn assumptions with large coverage that were close to actual sound assumptions, but theoretically unsound. Intuitively, in these cases, maximizing the coverage of the assumption, by searching for assumptions with a higher fitness, leads GP away from the generation of assumptions that are sound. For example, for the requirement (ϕ_1) of Two Tanks (TT), GP was returning, in one of its runs, the assumption $t1h \leq 0.5855 \vee t1h \geq 2.0065$ for one of the inputs of Two Tanks. This assumption is theoretically unsound. However, the assumption $t1h < 0.58 \vee t1h \geq 2$ is sound. For the same requirement, RS and DT were respectively returning, in one of their runs, the assumptions $t1h \geq 2.0259$ and $t1h < 0.5655 \vee t1h \geq 2.0265$ which are sound but have much lower coverage. Indeed, the assumption $t1h \leq 0.5855 \vee t1h \geq 2.0065$ has a larger coverage than $t1h \geq 2.0259$, since any input that satisfies $t1h \geq 2.0259$ also satisfies $t1h \geq 2.0065$. The assumption $t1h \leq 0.5855 \vee t1h \geq 2.0065$ also has a larger coverage than $t1h < 0.5655 \vee t1h \geq 2.0265$, since any input that satisfies $t1h < 0.5655$, also satisfies $t1h < 0.5855$, and any input that satisfies $t1h \geq 2.0265$ also satisfies $t1h \geq 2.0065$. Therefore, GP learned assumptions that are the closest to the one that has the largest coverage.

Considering each of the 32 combinations of requirements and input profiles separately, GP computed a sound assumption for 31 combinations in at least one of the 100 runs. DT and RS computed, respectively, a sound assumption for 26 and 22 of the 32 combinations in at least one of the 100 runs. Note that, computing a sound assumption once in 100 runs is still acceptable, since running our tool 100 times takes a few hours, when parallelization is used to execute different runs. So, in the worst case, engineers need to run our tool for a few hours to obtain assumptions, which is acceptable for our usage scenario. In the one case that all the three techniques failed to generate a sound assumption, all the generated assumptions were inconclusive. In the cases where only DT or RS could not compute a sound assumption, the assumption learned by GP has a complex structure and, therefore, could not be computed by DT and was difficult to be generated by RS. To statistically compare the distributions of the coverage values generated by GP with those generated by DT and RS, we used the Wilcoxon rank sum test [McD09] with the level of significance (α) set to 0.05. In both cases, the test rejected the null hypothesis (p-values < 0.05). Hence, assumptions learned by

Table 6.6: Number of runs, among the 100 runs of each requirement-profile combination, in which GP, DT and RS were able to compute a sound assumption

Req.	IP			IP'			IP''		
	GP	DT	RS	GP	DT	RS	GP	DT	RS
REG- ϕ_1	79	99	99	76	83	100	22	0	29
REG- ϕ_2	29	96	37	21	75	53	11	0	17
REG- ϕ_3	76	100	87	30	100	91	3	0	33
REG- ϕ_4	-	-	-	2	18	0	6	1	0
REG- ϕ_5	-	-	-	43	14	0	11	0	0
REG- ϕ_6	-	-	-	1	13	7	9	2	0
TU- ϕ_1	42	79	22	52	2	24	8	4	0
TU- ϕ_2	100	94	100	21	96	0	19	0	0
FSM- ϕ_1	-	-	-	-	-	-	100	86	100
FSM- ϕ_2	-	-	-	-	-	-	1	1	0
TT- ϕ_1	96	85	97						
TT- ϕ_2	93	62	89						
TT- ϕ_3	86	59	81						
TT- ϕ_4	93	55	93						
TT- ϕ_5	97	54	98						
TT- ϕ_6	92	55	89						
TT- ϕ_7	84	49	79						
TT- ϕ_8	88	94	85						
AC- ϕ_1	0	0	0						

* Symbol “-” marks entries related with input profiles for which the requirements are violated or satisfied.

For the eight requirements of Two Tanks (TT), only the input profile IP was considered since this model only supports constant input signals.

For AC, we considered the input profile IP suggested by our industrial partner.

GP have a significantly larger coverage than those learned by RS and DT.

The box plots in Figure 6.6 depict the behavior of GP, DT, and RS across the input profiles IP, IP' and, IP''. Each box plot reports the coverage value of one tool for a given input profile and is labelled with the percentage of cases, across the runs associated with that input profile, in which the tool was able to compute a sound assumption (value reported below the box plot). The results confirm that GP generates assumptions with a larger coverage than DT and RS, however, with a negligible loss of capacity to compute sound assumptions. For GP and DT, the test returned p-values lower than 0.05 for IP and IP'. For IP'', the p-value is greater than 0.05 (i.e., 0.1) since the sample size is too small to reject the null hypothesis. For GP and RS, the test returned p-values lower than 0.05 for all the input profiles. The results show that, as expected, the more complex the input profile, the more difficult the computation of a sound assumption. Therefore, to handle more complex input profiles, developers should tune the values of the parameters in Table 6.5, e.g., by increasing the timeout, the number of test cases, and the population size.

The answer to **RQ1** is that, on the considered study subjects, in contrast to DT and RS, GP can learn a sound assumption for 31 combinations out of 32 combinations of requirements and input profiles. The assumptions computed by GP have also a significantly larger coverage (20% and 8%) than those learned by DT and RS.

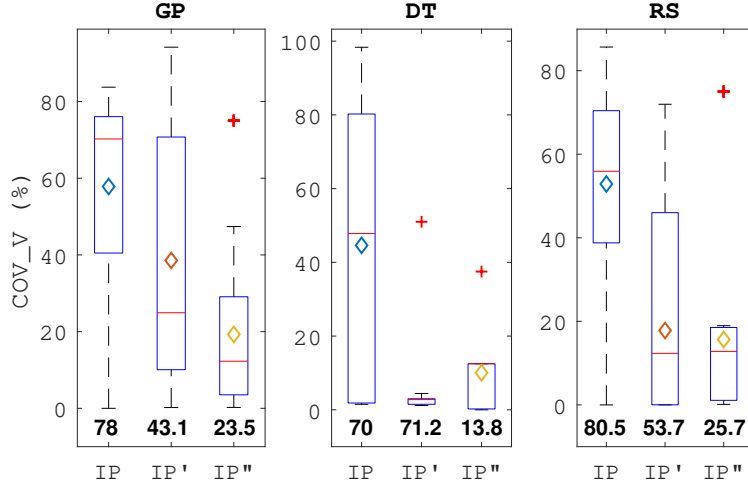


Figure 6.6: Comparing GP, DT, and RS. The box plots show the coverage value of GP, DT, and RS for the input profiles IP, IP' and, IP'' (labels on the bottom of the figure). Diamonds depict the average. The value below the box plot is the percentage of runs, across all the runs of each input profile, in which the technique was able to compute a sound assumption.

RQ2-1 — Usefulness of the assumptions with large coverage

To check whether GP can learn assumptions with large coverage similar to the one that would be manually defined by engineers, we empirically evaluated GP by considering the AC component of ESAIL. We analyzed the assumptions computed by GP in collaboration with the industrial CPS engineers that developed ESAIL. In this question, since we do not limit our analysis to sound assumptions, EPICuRus was configured to return a sound assumption, if found, or the assumption computed in the last iteration of EPICuRus, none of the assumptions generated across the different iterations could be proven to be sound.

Methodology and Experimental Setup. To learn assumptions on the 27 input signals of the ten inputs of ESAIL (see Table 6.1), we considered the parameter settings in Table 6.7. Cells marked with a gray background color denote parameter values that differ from the one considered for RQ1. We increased the values assigned to the test suite size (TS_Size) and the timeout (Timeout), since AC is significantly more complex than the other models. Recall from RQ1 that the parameter values in Table 6.5 did not lead to any sound assumption. The number of conjunctions (Max_Conj) and disjunctions (Max_Disj) are respectively set to 1 and 0 since, according to our industrial partner, the assumptions can be represented as a conjunction of two inequalities among complex arithmetic expressions (see Section 6.1). In practice, engineers do not know a priori the assumption with the largest coverage of the system. However, they can select parameter values based on their domain knowledge combined with experiments. The values assigned to Const_Min and Const_Max are, respectively, the lowest and the highest values the input signals of AC can assume. We considered the input profile IP with a single control point since, given the time domain (see Section 6.1) and according to the engineers of ESAIL, this input profile is sufficiently complex to represent changes in the inputs of AC over the considered time domain. We assumed the ranges $[-0.01, 0.01]$, $[-0.01, 0.01]$, $[0, 0.005]$, $[0, 0.005]$, and $[0, 0.005]$ as input domain for each of the input signals of the inputs q_t , q_e , ω_t , ω_e , and Rwh, respectively. We set the other inputs to constant values provided by our industrial

Table 6.7: Values for the parameters of Table 6.3 used to assess whether EPIcuRus can learn assumptions similar to the one manually defined by engineers.

	Parameter	Value	Parameter	Value
	SBA	GP	ST	$[0, 1]$ s
EP	TS_Size	3000	Stop_Crt	Timeout
	Timeout	5h	Nbr_Runs	100
	Max_Conj	1	Max_Disj	0
	Const_Min	-0.1	Const_Max	0.1
GP	Max_Depth	5	Init_Ratio	50%
	Pop_Size	500	Gen_Size	100
	Sel_Crt	TRS	T_Size	7
	Mut_Rate	0.1	Cross_Rate	0.9

Table 6.8: Minimum and maximum value of each term of exp.

Term	Min	Max	Term	Min	Max
T1	0	3.91	T2	-1.66	0
T3	0	0.175	T4	-0.0012	0
T5	-0.1187	0	T6	0	0.0897
T7	0	0.025	T8	-0.025	0
T9	-0.0013	0	T10	0	0.0013

partner.

We considered 100 runs of EPIcuRus and saved the assumptions computed for requirement ϕ_1 in each of these runs. Then, to assess whether EPIcuRus can learn assumptions similar to the ones that would be manually defined by engineers, we proceeded as follows. We elicited the assumption of AC for ϕ_1 in collaboration with the ESAIL engineers as described in Section 6.1. This was done by consulting the Simulink model design and the design documents of the satellite. We then compared the assumptions returned by EPIcuRus and the one we elicited in collaboration with the ESAIL engineers, i.e., the assumption A_1 for ϕ_1 . While eliciting this assumption, we found discrepancies between the model design and the design documents of the satellite. The problems were in the design documents of the satellite and were fixed by ESAIL engineers. We corrected our assumption accordingly to match the actual ESAIL design model. To assess the extent to which GP can learn sound assumptions similar to the ones that would be manually defined by engineers, we analyzed how many runs of EPIcuRus were able to learn each of the terms of the predicates $P_1(t)$ and $P_2(t)$. Note that, while the values of `Const_Min` and `Const_Max` are set to -0.1 and 0.1, coefficients of the terms with higher values (i.e., +783.3 in A_1) can be generated by selecting small values for the constant terms, i.e., +1 and -1 in the assumption A_1 . For example, our algorithm can generate the assumption $0.783 \cdot \omega_e \cdot x(t) < 0.001$ which is equivalent to $+783.3 \cdot \omega_e \cdot x(t) < 1$. We relied on classical arithmetic properties to scale up and down the values of the coefficients. Furthermore, given the domain of the inputs, the minimum and the maximum value for each term in exp is reported in Table 6.8. Given the ranges in the table, the term that can assume the highest value in exp is T1, followed by T2. For example, for term T2 the minimum value is -1.66 (i.e., $-332.6 \cdot 0.005$) and the maximum value is 0 (i.e., $-332.6 \cdot 0$).

6. COMBINING GENETIC PROGRAMMING AND MODEL CHECKING TO GENERATE ENVIRONMENT ASSUMPTIONS

Table 6.9: The value C of the coefficient of the term T_i in exp , The number N of runs in which EPIcuRus was able to learn T_i in P_1 and P_2 . The percentage S of runs in which the sign of T_i was correct. The average D and the maximum MaxD of the difference between the coefficient of T_i of A_1 and the one returned by EPIcuRus.

T_i	C	P_1				P_2			
		N	S	D	MaxD	N	S	D	MaxD
T1	783	59	71	679	4328	88	94	256	1550
T2	-332	69	91	247	1373	74	68	409	1331
T3	3	3	33	21	35	9	11	41	203
T4	-50	5	40	17682	87238	17	58	30778	242066
T5	-4751	1	0	4751	4751	3	33	4949	5345
T6	3588	1	0	3588	3588	4	25	3520	3998
T7	1000	0	-	-	0	0	-	-	0
T8	-1000	0	-	-	0	0	-	-	0
T9	-54	5	60	17536	87133	6	50	1585	4732
T10	54	0	-	-	0	1	0	54	54

Results. We obtained 100 assumptions from 100 runs of EPIcuRus. These assumptions could not be proven to be sound by QVtrace due to the complexity of their mathematical expressions. We analyzed and compared the syntax and the semantics of these assumptions with respect to the actual assumption of AC (A_1). The results are shown in Table 6.9. Specifically, Table 6.9 shows which of the terms T1, T2, ..., T10 of P_1 and P_2 , respectively, appear in the 100 assumptions computed by EPIcuRus. Each row of the table reports four metrics computed for one of the ten terms of both P_1 and P_2 . For each term T_i of P_1 or P_2 , the tables' columns show the following:

- **N-labelled columns.** The number of runs in which the assumptions generated by EPIcuRus contain the term T_i but not necessarily with the same coefficient as that appearing in the expression exp .
- **S-labelled columns.** The percentage of the runs in which the sign of T_i is the same as its sign in the expression exp over all the runs where the generated assumption by EPIcuRus contained T_i .
- **D-labelled columns.** The average of the differences between the values of the coefficients of T_i in exp and in the assumptions returned by EPIcuRus.
- **MaxD-labelled columns.** The maximum of the differences between the values of the coefficients of T_i in exp and in the assumptions returned by EPIcuRus.

The column labeled by C in Table 6.9 shows the values of the coefficients of the terms T_i in exp .

The results in Table 6.9 show that the terms T1 and T2 of P_1 and P_2 , that can yield the highest values among other terms (see Table 6.8), were contained in the assumptions returned by EPIcuRus in 59 and 69, and 88 and 74, out of the 100 runs, respectively. Note that GP learns assumptions with an arbitrary structure as it does not know the structure of the assumption a priori. For this reason, the terms T1 and T2 of P_1 are contained in a different number of assumptions than the terms T1 and T2 of P_2 , respectively. The other terms of A_1 , that yield negligible values compared with T1 and T2, cannot be effectively learned by EPIcuRus, i.e., they are contained in the assumptions returned by EPIcuRus in only a limited number of runs (< 17 each). This is an inherent property of the search as it cannot learn terms that are low. Note that, in most of the runs (78 out of 100), all the terms

learned by EPIcuRus are either part of P_1 or part of P_2 . Only in 28 runs the assumptions produced by EPIcuRus contained a spurious term. Furthermore, in these cases, given the input domains, the values the spurious terms could yield are irrelevant compared to the other terms. To conclude, since the terms T1 and T2 of P_1 and P_2 were contained in the assumptions returned by EPIcuRus in 59 and 69, and 88 and 74, out of the 100 runs, engineers are very likely to learn an assumption that contains the terms T1 and T2 by executing EPIcuRus a few time, which would take less than a day. For example, for the term T1 of P_1 the probability of finding it during the first, second, or third run is 0.9311 (i.e., $0.59 + (1 - 0.59) \cdot 0.59 + (1 - 0.59)^2 \cdot 0.59$). This shows that the term T1 of P_1 is likely to be computed within three runs.

When the terms T1 and T2 were contained in the assumptions computed by EPIcuRus, the sign was correct in 71% and 91%, and 94% and 68% of the cases, for P_1 and P_2 , respectively. The average of the differences between the coefficients of the terms T1 and T2 of P_1 and P_2 in A_1 and in the assumptions returned by EPIcuRus are 679 and 247, and 256 and 409, respectively. The maximum of the differences between the coefficients of the terms T1, and T2 of P_1 and P_2 in A_1 and in the assumptions returned by EPIcuRus are 4328 and 1373, and 1550 and 1331, respectively. Note that the coefficients of the terms T1, and T2 of P_1 and P_2 in A_1 are 783 and -332 , respectively. As we can see, the coefficient values computed by EPIcuRus are not too different from the actual coefficient values for T1, and T2, even though EPIcuRus could technically select any arbitrary number as a coefficient for these terms. However, provided with such a large search space, EPIcuRus has been able to select coefficients for these terms that are in the same order of magnitude (i.e., number's nearest power of ten) as their actual coefficients. Such accuracy for coefficient estimates is acceptable when the coverage of the assumptions is prioritized and what matters is the identification by engineers of the assumptions' terms that yield the highest values among other terms in the actual assumption. A higher accuracy would not have significant practical benefits since the model checker would anyway not be able to confirm the soundness of the assumption.

To illustrate how useful EPIcuRus can be in practice, let us take three of the 100 runs for which EPIcuRus returned the following representative assumptions:

$$\begin{aligned}
A_{r1}(t) &= \underbrace{+1003.4 \cdot \omega_e_x(t)}_{P1-T1} \underbrace{-515.8 \cdot \omega_e_y(t)}_{P1-T2} + 1 \geq 0 \wedge \\
&\quad \underbrace{958.0 \cdot \omega_e_x(t)}_{P2-T1} \underbrace{-452.8 \cdot \omega_e_y(t)}_{P2-T2} - 1 \leq 0 \\
A_{r2}(t) &= \underbrace{+757.2 \cdot \omega_e_x(t)}_{P2-T1} \underbrace{-413.1 \cdot \omega_e_y(t)}_{P2-T2} \\
&\quad \underbrace{-4787.4 \cdot \omega_e_x(t) \cdot \omega_e_y(t)}_{P2-T4} \\
&\quad \underbrace{-4787.4 \cdot \omega_e_y(t)^2}_{P2-T9} - 1 \leq 0 \\
A_{r3}(t) &= \underbrace{+757.6 \cdot \omega_e_x(t)}_{P1-T1} \underbrace{-448.4 \cdot \omega_e_y(t)}_{P1-T2} \\
&\quad + 448.4 \cdot \omega_e_x(t)^2 + 1 \geq 0 \wedge \\
&\quad \underbrace{856.8 \cdot \omega_e_x(t)}_{P2-T1} \underbrace{-419.8 \cdot \omega_e_y(t)}_{P2-T2} \\
&\quad - 8.6 \cdot \text{Rwh_z}(t) - 1 \leq 0
\end{aligned}$$

where P1-T1, P1-T2, ... and P2-T1, P2-T2, ... label the terms T1, T2, ... of $P_1(t)$ and $P_2(t)$, respectively. EPIcuRus returns both assumptions that contain the terms T1 and T2 of both predicates (e.g., $A_{r1}(t)$), and assumptions that contain the terms T1 and T2 of only one of the predicates (e.g.,

$A_{r_2}(t)$). Some assumptions contain only terms that are part of A_1 (e.g., $A_{r_1}(t)$, $A_{r_2}(t)$), others contain additional spurious terms (e.g., $A_{r_3}(t)$), i.e., terms that are not part of A_1 (e.g., $8.6 \cdot \text{Rwh_z}(t)$). Finally, note that, when EPICuRus learns a term present in both $P_1(t)$ and $P_2(t)$ (e.g., T_1), the coefficients of P_1-T_1 and P_1-T_2 are not necessarily equal.

Discussion. When EPICuRus is configured to learn assumptions with large coverage that are not necessarily proven to be sound, our results show that the resulting assumptions include the terms that yield the highest values among other terms in the actual assumption. Even though, in the generated assumptions, not all the terms are present, and the values of their coefficients are only estimates, ESAIL engineers confirmed that these assumptions are still useful and beneficial for designing CPS components, because they can help engineers identify flaws in their components.

Specifically, engineers generally know which input signals yield the highest values and expect to see those input signal variables in the assumptions generated by EPICuRus. The absence of those variables in the generated assumptions may indicate flaws. For example, by analyzing $A_{r_1}(t)$, engineers understand that estimated speeds of the satellite across x and y axes have a significant impact on the satisfaction of ϕ_1 , as expected. Furthermore, the assumption also provides high level and approximate information regarding the values of $\omega_{e_x}(t)$ and $\omega_{e_y}(t)$ that satisfy the requirement.

Engineers can execute several runs of EPICuRus and obtain a report containing the information shown in Table 6.9. By consulting the data reported in the table, they can understand which terms are present in most of the assumptions computed by EPICuRus and yield the highest values. Note that, the term of the assumption yielding the highest value is likely to be the one that has the highest impact on the property satisfaction. Running 100 runs of EPICuRus requires approximately 20 days. However, the results can be obtained within approximately one day by running 20 instances of EPICuRus in parallel. This is a reasonable solution for computing assumptions of critical CPS components.

The assumptions returned by EPICuRus could not be learned by our previous version of EPICuRus that relies on DT to learn assumptions. Finally, EPICuRus is the only existing tool that is able to synthesize assumptions for CPS Simulink components. Therefore, there is no alternative that engineers could consider to address their need.

The answer to **RQ2-1** is that, among the terms of the assumptions identified by the LuxSpace engineers, EPICuRus configured with GP was able to learn the terms that yield the highest values among other terms in the actual assumption. These assumptions could not be learned with any other tool.

RQ2-2 — Usefulness of the Sound Assumptions

To check whether GP can learn sound assumptions similar to the one that would be manually defined by engineers, we configured EPICuRus to increase the chances of computing simpler assumptions whose soundness can be verified by QVtrace.

Methodology and Experimental Setup. To check whether learning simpler assumptions allows QVtrace to prove that they are sound, we configured EPICuRus using the values of the parameters in Table 6.10. Compared with the values in Table 6.7, we decreased the timeout from five hours to one hour, the number of generations (Gen_Size) from 100 to 10, and set the values assigned to Const_Min and Const_Max to -0.001 and 0.001 , respectively. We considered 100 runs of EPICuRus and saved the assumptions computed for the requirement ϕ_1 in each of these runs. Then, we computed the percentage of the runs in which EPICuRus was able to compute a sound assumption.

Results. Across the 100 runs, EPICuRus was able to compute sound assumptions in 16 runs. On average, generating one sound assumption for AC took about 6 hours. All the 16 sound assumptions generated by EPICuRus have lower coverage (simpler) than A_1 , the actual assumption of AC. We identified three distinct patterns to categorize the 16 generated assumptions. Below, we list the patterns and, for each one, we show which terms from the original assumption A_1 appear in the pattern.

$$\begin{aligned}
 A_{s1}(t) &= \underbrace{a \cdot \omega_e_y(t)}_{P1-T2} - c \leq 0 \wedge \\
 &\quad \underbrace{b \cdot \omega_e_x(t)}_{P2-T1} - d \leq 0 \\
 A_{s2}(t) &= \underbrace{a \cdot \omega_e_x(t)}_{P2-T1} + \underbrace{b \cdot \omega_e_y(t)}_{P2-T2} - c \leq 0 \\
 A_{s3}(t) &= \underbrace{a \cdot \omega_e_x(t)}_{P2-T1} + \underbrace{b \cdot \omega_e_y(t)}_{P2-T2} + \underbrace{c \cdot \omega_e_z(t)}_{P2-T3} \leq 0
 \end{aligned}$$

The above assumptions, although simpler than the original assumption A_1 , have sufficiently large coverage as confirmed by ESAIL engineers. For example, the $A_{s1}(t)$ pattern indicates that, when the values of the estimated speed of the satellite along its x and y axes are low, requirement ϕ_1 is satisfied regardless of the values assigned to the other inputs. Similarly, the $A_{s2}(t)$ and $A_{s3}(t)$ patterns indicate that, when the sum of the speeds of the satellite along its x and y axes is low, requirement ϕ_1 is satisfied. In other words, despite being an oversimplification of reality, learned assumptions provide correct insights into the conditions leading to the satisfaction of requirements.

To verify that the satisfaction of the assumption generated by EPICuRus entails the satisfaction of the baseline assumption, we compared the assumptions returned by EPICuRus with the baseline assumption A_1 of AC (see Section 6.4) using QVtrace. We loaded a Simulink model representing the baseline assumption in QVtrace. The inputs of the model are the input signals of the assumptions. The output is a Boolean value: *true* if the assumption is satisfied, *false* otherwise. Then, we iteratively loaded each of the assumptions generated by EPICuRus in QVtrace. For all the assumptions generated by EPICuRus, QVtrace confirmed that the output of the Simulink model is *true* for all the inputs that satisfy the assumption. This shows that the satisfaction of the assumption generated by EPICuRus entails the satisfaction of the baseline assumption. Therefore, any component that satisfies the assumption generated by EPICuRus also satisfies the actual (baseline) assumption of the AC component.

The answer to **RQ2-2** is that, when EPICuRus was configured with parameters that lead to the generation of sound assumptions, EPICuRus was able to generate sound assumptions in 16 runs out of 100. Therefore, through multiple runs, EPICuRus can generate a sound assumption within approximately six hours. Though simpler than actual assumptions, these learned assumptions appear to provide correct and useful insights.

Discussion

In the following, we discuss (i) the impact of our results on the documentation practices of LuxSpace, and (ii) the impact of the design choices of the components of EPICuRus and the configuration settings we selected on the obtained results.

Documentation Practices. LuxSpace engineers detail the behavior of ESAIL in a design document. The design document is divided into several sections, one for every component of ESAIL. For every component, among various information, the specification document contains (i) the description

Table 6.10: Values for the parameters of Table 6.3 used for RQ2-2.

	Parameter	Value	Parameter	Value
EP	SBA	GP	ST	[0, 1]s
	TS_Size	3000	Stop_Crt	Timeout
	Timeout	1h	Nbr_Runs	100
GP	Max_Conj	1	Max_Disj	0
	Const_Min	-0.001	Const_Max	0.001
	Max_Depth	5	Init_Ratio	50%
	Pop_Size	500	Gen_Size	10
	Sel_Crt	TRS	T_Size	7
	Mut_Rate	0.1	Cross_Rate	0.9

of its inputs and outputs; (ii) the mathematical formulae that define the behavior of the component; (iii) the natural language explanation of these formulae; and (iv) a discussion of the design decisions made by the engineers to define the behavior of the component. The above information was also present in the specification documents of the benchmark models [NGM⁺19] of Lockheed Martin [loc20] we considered in Section 6.1. Neither LuxSpace nor Lockheed Martin engineers included the assumptions of the software components in the specification documents. The ESAIL Simulink model contains a set of assertion blocks [ass21] encoding simple component assumptions that check whether the values assumed by some of the signals are included within valid ranges. For the Lockheed Martin models, the input type (e.g., Boolean or Real) was described in the specification document, but the valid input ranges were not specified. None of the models included complex assumptions containing arithmetic expressions defined over multiple variables, such as assumption A_1 presented in Section 6.1. This confirms that manually identifying assumptions is difficult, especially when the component has many inputs and its behavior is defined by complex and non-linear equations. As confirmed by the results reported in our evaluation, EPICuRus helps engineers in addressing this problem by automatically identifying complex and sound component assumptions.

Fitness Measure. Our fitness measure guides the search toward sound assumptions with large coverage. Coverage as the core of the fitness function might lead to assumptions that, accidentally and unnecessarily, correlate signals generated by different components. However, in our evaluation, this was not the case when soundness was a priority (see RQ2-2). When soundness was not a priority only in 28 runs (over 100) the assumptions produced by EPICuRus contained a spurious term. Furthermore, in these cases, given the input domains, the values the spurious terms could yield are negligible compared to the other terms. Therefore, based on our results, the number of assumptions that, accidentally and unnecessarily, correlate with signals generated by different components is limited and only concern the case in which soundness was not a priority.

Metric to Compute the Cardinalities of the Valid Input Sets. To compute the size of the valid input sets we decided not to prioritize any of the input dimensions (e.g., speed, distance, angle). This was done both in our search-based algorithm (see Section 6.3) and in our evaluation (see Section 6.4). This is because we wanted to maximize coverage across all input dimensions equally, regardless of their types. The effectiveness of our metrics is confirmed by the results obtained for RQ2-1 and RQ2-2. If there is a need to focus coverage on certain input dimensions, engineers can modify these metrics, e.g., by giving a higher weight to some of the input types.

Control Points-Based to Signal-Based Assumptions. To apply model checking on the generated assumptions, we translated control-point-based assumptions to signal-based ones. Our translation

is provided for a case where assumptions on control points are forced to hold continuously across the time interval between the control point and its preceding control point since it generates simpler assumptions that (a) are easier to be understood by engineers and (b) are more likely to lead to a true or false verdict when analyzed by the model checker. Our evaluation showed that, our strategy successfully learnt sound assumptions with high coverage for all of our study subjects. However, more complex translations that encode correlations of signals in which the expression needs a time shift on the time series can be used.

Configuration Settings. Our fitness function guides the search through the sound assumptions that ensure a high coverage. Therefore, our fitness function learns constraints on the terms of the assumption A_1 that can assume the highest value. As mentioned in Section 6.4. When soundness is not a priority, in 28 runs (over 100) the assumptions produced by EPIcuRus contained a spurious term. The configuration settings of EPIcuRus (see Table 6.7) influence the number of spurious terms learned by our tool. The higher the timeout (`Timeout`), the more likely is EPIcuRus to produce spurious terms since it performs more iterations, and therefore can learn assumptions that have a larger coverage (which may contain unnecessary correlation between input signals). The higher the `Max_Dept`, the higher is the chance to have spurious terms in the assumption, since the tool can learn bigger assumptions.

Threats To Validity

The set of models we selected for the evaluation and their features influence the generalizability of our results. Related to this thread, we note that: First, the public domain benchmark of Simulink models we used in our study have been previously used in the literature on testing of CPS models [NGM⁺19, MNGB19]; second, the models in the benchmark represent realistic and representative models of CPS components from different domains; third, our industry satellite model represents a realistic and representative CPS model for which we could develop assumptions manually by collaborating with the engineers who had developed those models; fourth, our results can be further generalized by additional experiments with diverse types of CPS components and by assessing EPIcuRus over those components.

6.5 Related Works

In this section, we provide the related works of the work presented in Chapter 5 and Chapter 6. This section compares our approach EPIcuRus to the existing works by discussing the differences and similarities regarding the verification, testing and monitoring of CPS and regarding the assumptions inference for software components.

Verification, Testing and Monitoring of CPS. Approaches to verifying, testing, and monitoring CPS were proposed in the literature e.g., [MNGB19, MBG⁺20c, ALFS11b, MNBP20, FGD⁺11, Tiw12, BMB⁺20, MVBB21, BAN⁺20, MAES19, AWM⁺17, SCN⁺20]). However, these approaches usually assume that the assumptions on the inputs of the CPS component under analysis are already specified. Those approaches verify, test, and monitor the behavior of the CPS component for the input signals that satisfy those assumptions. Our work is complementary to those and, in contrast, it automatically identifies (implicit) assumptions on test inputs. Considering those assumptions is an important pre-requisite to ensure that testing and verification results are not overly pessimistic or spurious [CGP03, BG03, GPC04, DLS12, HMZ12, GPB08].

Compositional reasoning. Assume-guarantee and design by contract approaches were proposed in the literature to support hardware and software verification (e.g., [AAS20a, DLTT13, SVDP12, MSCG19, MSCC18, dAH01a, HQR98, dAH01b, BOV⁺19, AAS20b]). Assume-guarantee contracts rep-

represent the assumptions a CPS component makes about its environment, and the properties it satisfies when these assumptions hold, i.e., its guarantees. Some recent work discusses how to apply assume-guarantee to signal-based modeling formalisms, such as Simulink and analog circuits (e.g., [SNWS09, NXO⁺14, NFIS14]). However, these frameworks assume that assumptions and guarantees are manually defined by the designers of the CPS components. Our work is complementary as the assumptions learned by EPIcuRus can be used within these existing frameworks. Finally, our work also differs from assume-guarantee testing, where the assumptions defined during software design are used to test the individual components of the system [GPB08].

Learning Assumptions. The problem of automatically inferring assumptions of software components, a.k.a supervisory control problem, was widely studied in the literature (e.g., [CGP03, CL08, GPB02, MRS19, BG03, GPC04, RW87, RWW89, CAG20, PKB⁺20]). However, the solutions proposed in the literature are solely focused on components specified in finite-state machines and are not applicable to signal-based formalisms (e.g., Simulink models), that are widely used to specify CPS components.

Kampmann et al. [KHSZ20] proposed an approach to automatically determine under which circumstances a particular program behavior, such as a failure, takes place. However, this approach uses a decision tree learner to observe and learn which input features are associated with the particular program behavior under analysis. As such, for our usage scenario, this approach is going to inherit the same limitations of our earlier version of EPIcuRus.

Dynamic invariants generators (e.g., [EPG⁺07a, CPS16]) infer conditions that hold at certain points of a program. They generate a set of candidate invariants and return the best candidates that hold over the observed program executions. In contrast, the goal of this work is to generate environment assumptions for signal-based modeling formalisms, such as Simulink models. Environment assumptions can be considered as a specific type of invariant, but introduce specific challenges, such as the ones considered in this work.

Property inference aims at automatically detecting properties that hold in a given system. It was also recently applied to feed-forward neural network [GCPT19]. While many approaches for property inference were proposed in the literature (e.g., [FL01, EPG⁺07b, GLMN14, GNMR16]), they do not consider signal-based modeling formalisms (e.g., Simulink) which are the targets of this work.

Template-based specification mining are used to synthesize assumptions following with a certain structure [LDS11, AMT13]. However, solutions from the literature (e.g., [LDS11, AMT13, KBD⁺20]) use LTL-GR(1) to express assumptions. These formalisms are substantially different and less expressive than the one considered in this work, and can not express the signal-based assumptions generated by EPIcuRus.

In this work, we combined model checking and model testing to learn assumptions. This idea was supported by a recent study [NGM⁺19] that analyzed the complementarity between model testing and model checking for fault detection purposes.

This chapter significantly extends our previous version of EPIcuRus [GMN⁺20]. Our extension enables learning assumptions containing conditions defined over multiple signals related by both arithmetic and relational operators. Assumptions containing conditions defined over multiple signals related by both arithmetic and relational operators are common for industrial CPS components. This is confirmed by our industrial case study from the satellite domain. Differently than our previous work, we used genetic programming to synthesize complex assumptions of CPS components. Finally, we performed an extensive and thorough evaluation of the assumptions computed by the extended version of EPIcuRus using an industrial case study from the satellite domain. The assumptions computed by EPIcuRus were evaluated in collaboration with the engineers that developed the satellite.

Learning Parameters. The problem of learning (requirement) parameters from simulations was

extensively studied in the literature [JDDS15, JDVDAS13, BdLN11]. Our work is significantly different from those, since it aims to learn assumptions on the input signals.

EPIcuRus extends counterexample-guided inductive synthesis [STB⁺06] by exhaustively verifying the learned assumptions using an SMT-based model checker. Furthermore, differently from counterexample-guided inductive synthesis, EPIcuRus (i) targets signal-based formalisms, that are widely used in the CPS industry, (ii) extracts assumptions from test data, and (iii) uses test cases to efficiently produce a large amount of data to be fed in our machine learning algorithm to derive sound assumptions with large coverage.

6.6 Conclusion

This chapter proposes a technique to learn complex assumptions for CPS systems and components. Our technique uses genetic programming (GP) to learn assumptions containing conditions defined over multiple signals related by both arithmetic and relational operators. Environment assumptions are required to ensure that the CPS under analysis meets its requirements and to avoid spurious failures during the verification process. We evaluated our approach using 12 models of CPS components with 94 requirements which includes 11 models from Lockheed Martin [loc20] and the model of the attitude control component of ESAIL with four requirements provided by LuxSpace [Lux19]. Our evaluation shows that our approach can learn many more sound environment assumptions compared to the alternative baseline techniques. Further, our approach is able to learn assumptions that have a significantly larger coverage than those generated by existing techniques. Finally, for our industrial CPS model, our approach is able to generate assumptions that are sufficiently close to the assumptions manually developed by engineers to be of practical value. In the next chapter, we summarize the challenges that are addressed and the contributions that are achieved in this dissertation. We further provide different ways to possibly extend the presented works.

Chapter 7

Conclusions & Future Work

In this chapter, we summarize the contributions of this dissertation and we discuss some perspectives on the potential future work in this area.

7.1 Summary

We focus on the problem of the verification of behavioral design models of CPS specified in Simulink. Cyber-physical systems are joint dynamics of computers, software, networks, and physical processes. This complex design is the root to certain problems that emerge during the verification of these systems. One of the main challenges is to guarantee the correct functioning of the components that are computationally intensive and hence require time to perform tasks like measuring and controlling the dynamics of the system. Despite the extensive domain expertise, traditional verification practices can be erroneous and sometimes undecidable on complex systems.

In this dissertation, we propose several approaches to alleviate the above challenges. We addressed the problem of expensive test executions of the systems under analysis by applying online checking which stops as soon as a failure is detected, while we handle time-continuous CPS behaviors and uncertainties due to CPS interactions with the environment. Moreover, we evaluated the capabilities of existing verification strategies and we provided lessons learned as well as guidelines regarding the strengths and weaknesses of the different techniques when applied to different types of CPS models. Further, we used a combination of search-based testing, machine learning and model checking to synthesize complex environment assumptions under which the CPS components are guaranteed to satisfy given requirements. The work presented in this dissertation has been done in collaboration with LuxSpace [Lux19], a leading provider of space systems, applications and services in Luxembourg, and QRA Corp [qra19], a verification tool vendor to the aerospace, automotive and defence sectors in Canada.

Chapter 3 introduces SOCRaTes, an automated approach to generate online test oracles in Simulink. Our approach generates oracles that handle CPS Simulink models with continuous and uncertain behaviors. The oracles return a quantitative degree of satisfaction for each test input, which measures how far the test input is from satisfying or violating a given requirement. We evaluated SOCRaTes using 11 industry case studies. Our results show that (i) our requirements language is expressive enough to capture all the 98 requirements of our case studies; (ii) the effort

required by SOCRaTes to generate online oracles in Simulink is acceptable; and (iii) for large models, the online checking dramatically reduces the test execution time.

Chapter 4 performs an empirical evaluation and comparison of the capabilities of model checking and model testing techniques for finding requirements violations in Simulink models. This chapter presents an industrial Simulink model benchmark to evaluate the two main-stream verification techniques. Our results show that our model checking technique is effective and efficient in proving correctness of requirements on Simulink models that represent CPS components. However, this technique may fail to handle systems with complex dynamic behaviours. Contrary to model checking, the results of applying our model testing technique show that it can scale to large and complex CPS models. However, this technique is computationally expensive and lacks insights on the optimal search heuristics to be applied. Most importantly, combining the two techniques is the best way ahead to make use of the strengths and to cope with the limitations of these two main-stream automated verification techniques.

Chapter 5 introduces 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 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 areas with the largest coverage in the search space. EPIcuRus is applicable to complex signal-based modeling notations (e.g., Simulink) commonly used in cyber-physical systems. Our evaluation shows that EPIcuRus is able to infer assumptions for all the requirements in our case 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.

Chapter 6 extends the approach of EPIcuRus to learn more complex assumptions. This chapter proposes a technique to learn complex assumptions for CPS systems and components. Our technique uses genetic programming (GP) to learn assumptions over multiple signals related by both arithmetic and relational operators. Environment assumptions are needed to ensure that the CPS under analysis meets its requirements and to avoid spurious failures during verification. We evaluated our approach using 12 models of CPS components with 94 requirements provided by Lockheed Martin [loc20] and the model of the attitude control component of ESAIL with four requirements provided by LuxSpace [Lux19]. Our evaluation shows that our approach can learn many more sound environment assumptions compared to the alternative baseline techniques. Further, our approach is able to learn assumptions that have a significantly larger coverage than those generated by existing techniques. Finally, for our industrial CPS model, our approach is able to generate assumptions that are sufficiently close to the assumptions manually developed by engineers to be of practical value.

7.2 Future Work

In the future, we would like to further assess the generalizability of our solutions (Chapter 3 to Chapter 6). We plan to evaluate the applicability and effectiveness of our approaches on a larger benchmark which involves a higher number of realistic models and requirements that are representative of different domains of CPS. In fact, the selection of the models used in this dissertation and the features contained in those models, influence the extent to which our results can be generalized. While the benchmark models, introduced in Chapter 3, Section 3.3 and Chapter 4, Section 4.1 represent realistic models of CPS systems from different domains, our results can be further generalized by adding experiments with diverse types of systems and by assessing EPIcuRus over those systems.

Our industry microsatellite model can also be included in future experiments where we consider a larger set of requirements of the ESAIL model. Furthermore, our solutions are potentially useful in other cyber-physical domains, e.g., self-driving systems. Therefore, we plan to consider well-known industrial cases from different domains to prove the usefulness of our proposed approaches.

In Chapter 5, the evaluation confirms that the conjectures we defined hold and that the effectiveness of EPIcuRus is adequate for practical usages. In the following we discuss the practical implications of EPIcuRus.

1. EPIcuRus learns classification trees and converts them into classification rules. Directly learning classification rules [Mol19b] 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 [WFHP16], to generate them, and further, our solution based on classification trees already works reasonably well.
2. Decision trees applied in Chapter 5, Section 5.2 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). Therefore, we extended our approach in Chapter 6 to infer more complex assumptions using genetic programming. Alternatively, we can use other machine learning techniques (e.g., more expressive rules or clustering [WFHP16]).
3. EPIcuRus generates assumptions that assume that consecutive control points are connected using a linear interpolation function. Our evaluation shows that this assumption provides a good compromise between simplicity and realism. In order to make our approach more generalizable, we can expand it to consider more complex interpolation functions in particular when we have specific domain knowledge about the shapes of different input signals.

Our work can be extended in many different ways, a few of them are summarized in the following:

1. Our approach considers assumptions represented as a disjunction of one or more constraints defined as in Section 6.2. Although our results show that, for our case studies, EPIcuRus can learn sound assumptions with high coverage expressed in this form, learning assumptions of different forms can further increase the applicability of EPIcuRus. This would require extending the EPIcuRus *test generation* and *assumption generation* procedures and selecting a *model checker* that supports new forms of assumptions;
2. As discussed in Section 6.2 of the chapter, we did not use the interpolation function, specified within the input profile, to translate control points-based to signal-based assumptions. Analyzing and defining more complex translations is part of our future work;
3. Alternative techniques, such as program synthesis [Kit09, GPS⁺17], or Support Vector Machines [BL02] can be used for implementing the assumption generation step. These techniques can further improve the effectiveness of EPIcuRus;
4. EPIcuRus uses a fixed set of test cases to evaluate assumptions. Search-based software testing (SBST) is an alternative technique, which requires performing additional computations, that can be used for evaluating the generated assumptions. We plan to extend EPIcuRus to support the usage of SBST to evaluate the generated assumptions, and to assess whether using SBST is beneficial in practical applications given the additional computational cost;

5. The sanity check and soundness check steps use model-checking to determine if the given requirement is satisfied or violated by the model inputs satisfying the assumption. When the requirement is not satisfied, we plan to use the counterexample to guide the test generation step.

Finally, the assumptions generated by EPICuRus have many other potential usage scenarios, such as compositional reasoning, or supporting the detection of design flaws. Assessing how these assumptions support such usage scenarios is part of our future work.

Bibliography

- [AAI⁺21] Aitor Arrieta, Jon Ayerdi, Miren Illarramendi, Aitor Agirre, Goiuria Sagardui, and Maite Arratibel. Using machine learning to build test oracles: an industrial case study on elevators dispatching algorithms. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 30–39. IEEE, 2021.
- [AAS20a] A. Arrieta, J. A. Agirre, and G. Sagardui. A tool for the automatic generation of test cases and oracles for simulation models based on functional requirements. In *Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–5. IEEE, 2020.
- [AAS20b] Aitor Arrieta, Joseba Andoni Agirre, and Goiuria Sagardui. Seeding strategies for multi-objective test case selection: An application on simulation-based testing. In *Genetic and Evolutionary Computation Conference (GECCO)*, page 1222–1231. ACM, 2020.
- [AB11] Andrea Arcuri and Lionel C. Briand. Adaptive random testing: an illusion of effectiveness? In *International Symposium on Software Testing and Analysis, ISSA*, pages 265–275. ACM, 2011.
- [ACH⁺95] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A Henzinger, P-H Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Theoretical computer science*, 138(1):3–34, 1995.
- [Acq18] Paul Acquatella. Fast slew maneuvers for the high-torque-wheels biros satellite. *Transactions of the Japan Society for Aeronautical and Space Sciences*, 61(2):79–86, 2018.
- [Adm09] United States. Federal Aviation Administration. *Advanced Avionics Handbook*. FAA Handbooks Series. Aviation Supplies & Academics, Incorporated, 2009. URL: <https://books.google.lu/books?id=2xGuPwAACAAJ>.
- [AF13] Andrea Arcuri and Gordon Fraser. Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empirical Software Engineering*, 18(3):594–623, 2013.
- [AFS⁺13] Houssam Abbas, Georgios E. Fainekos, Sriram Sankaranarayanan, Franjo Ivancic, and Aarti Gupta. Probabilistic temporal logic falsification of cyber-physical systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(2s):95:1–95:30, 2013.

- [AIB12] Andrea Arcuri, Muhammad Zohaib Z. Iqbal, and Lionel C. Briand. Random testing: Theoretical results and practical implications. *IEEE Trans. Software Eng.*, 38(2):258–277, 2012. [doi:10.1109/TSE.2011.121](https://doi.org/10.1109/TSE.2011.121).
- [AIER15] Dejanira Araiza-Illan, Kerstin Eder, and Arthur Richards. Verification of control systems implemented in simulink with assertion checks and theorem proving: A case study. In *2015 European Control Conference (ECC)*, pages 2670–2675. IEEE, 2015.
- [AKLP10] Alessandro Abate, Joost-Pieter Katoen, John Lygeros, and Maria Prandini. Approximate model checking of stochastic hybrid systems. *European Journal of Control*, 16(6):624–641, 2010.
- [ALFS11a] Yashwanth Annapureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. S-taliro: A tool for temporal logic falsification for hybrid systems. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Part of the Joint European Conferences on Theory and Practice of Software*, Berlin, Heidelberg, 2011. Springer-Verlag.
- [ALFS11b] Yashwanth Annapureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. S-TaLiRo: A tool for temporal logic falsification for hybrid systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 254–257. Springer, 2011.
- [Alu11] R. Alur. Formal verification of hybrid systems. In *International Conference on Embedded Software (EMSOFT)*, pages 273–278, 2011.
- [Alu15] Rajeev Alur. *Principles of Cyber-Physical Systems*. MIT Press, 2015.
- [AMN12] César Andrés, Mercedes G. Merayo, and Manuel Núñez. Formal passive testing of timed systems: theory and tools. *Software Testing, Verification and Reliability*, 22(6):365–405, 2012. [doi:10.1002/stvr.1464](https://doi.org/10.1002/stvr.1464).
- [AMT13] Rajeev Alur, Salar Moarref, and Ufuk Topcu. Counter-strategy guided refinement of GR(1) temporal logic specifications. In *Formal Methods in Computer-Aided Design, FMCAD*, pages 26–33. IEEE, 2013.
- [ANBS16] Raja Ben Abdesslem, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. Testing advanced driver assistance systems using multi-objective search and neural networks. In *International Conference on Automated Software Engineering (ASE)*. ACM, 2016.
- [ass21] assertionblock, 2021. URL: <https://nl.mathworks.com/help/simulink/slref/assertion.html>.
- [Atk08] Kendall E Atkinson. *An introduction to numerical analysis*. John Wiley & Sons, 2008.
- [AWM⁺17] A. Arrieta, S. Wang, U. Markiegi, G. Sagardui, and L. Etxeberria. Search-based test case generation for cyber-physical systems. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 688–697. IEEE, 2017.
- [AWM⁺19] Aitor Arrieta, Shuai Wang, Urtzi Markiegi, Ainhoa Arruabarrena, Leire Etxeberria, and Goiuria Sagardui. Pareto efficient multi-objective black-box test case selection for simulation-based testing. *Information and Software Technology*, 114:137–154, 2019.

- [BAN⁺20] Markus Borg, Raja Ben Abdesslem, Shiva Nejati, Francois-Xavier Jegeden, and Donghwan Shin. Digital twins are not monozygotic-cross-replicating adas testing in two industry-grade automotive simulators. *arXiv preprint arXiv:2012.06822*, 2020.
- [BB12] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012.
- [BBB⁺12a] Jiri Barnat, Lubo Brim, Jan Beran, Ítalo R Oliveira, et al. Executing model checking counterexamples in simulink. In *2012 Sixth International Symposium on Theoretical Aspects of Software Engineering*, pages 245–248. IEEE, 2012.
- [BBB⁺12b] Jiri Barnat, Lubos Brim, Jan Beran, Tomas Kratochvila, and Italo R Oliveira. Executing model checking counterexamples in Simulink. In *TASE 2012*, pages 245–248. IEEE, 2012.
- [BBČ⁺06] Jiří Barnat, Luboš Brim, Ivana Černá, Pavel Moravec, Petr Ročkai, and Pavel Šimeček. Divine—a tool for distributed verification. In *International Conference on Computer Aided Verification*, pages 278–281. Springer, 2006.
- [BDD⁺18] Ezio Bartocci, Jyotirmoy Deshmukh, Alexandre Donzé, Georgios Fainekos, Oded Maler, Dejan Ničković, and Sriram Sankaranarayanan. Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications. In *Lectures on Runtime Verification*, pages 135–175. Springer, 2018.
- [BdLN11] R. V. Borges, A. d’Avila Garcez, L. C. Lamb, and B. Nuseibeh. Learning to adapt requirements specifications of evolving systems. In *International Conference on Software Engineering (ICSE)*. IEEE, 2011.
- [BDN17] Luciano Baresi, Marcio Delamaro, and Paulo Nardi. Test oracles for simulink-like models. *Automated Software Engineering*, 24(2):369–391, 2017.
- [BDNCT17] Alessio Balsini, Marco Di Natale, Marco Celia, and Vassilios Tsachouridis. Generation of simulink monitors for control applications from formal requirements. In *Industrial Embedded Systems (SIES)*, pages 1–9. IEEE, 2017.
- [BFHN18a] Alexey Bakhirkin, Thomas Ferrère, Thomas A Henzinger, and Dejan Ničković. The first-order logic of signals: keynote. In *International Conference on Embedded Software*, page 1. IEEE Press, 2018.
- [BFHN18b] Alexey Bakhirkin, Thomas Ferrère, Thomas A. Henzinger, and Dejan Ničković. The first-order logic of signals: Keynote. In *International Conference on Embedded Software*, EMSOFT, pages 1:1–1:10. IEEE, 2018. URL: <http://dl.acm.org/citation.cfm?id=3283535.3283536>.
- [BFOS84] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984.
- [BG03] Howard Barringer and Dimitra Giannakopoulou. Proof rules for automated compositional verification through learning. In *In Proc. SAVCBS Workshop*, pages 14–21, 2003.
- [BG11a] R Baheti and H Gill. The impact of control technology: Cyber-physical systems. *IEEE Control Systems Society*, pages 1–6, 2011.

- [BG11b] Radhakisan Baheti and Helen Gill. Cyber-physical systems. *The impact of control technology*, 12(1):161–166, 2011.
- [BHM⁺14] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2014.
- [BHM⁺15] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *Transactions on Software Engineering*, 41(5):507–525, 2015.
- [BHSL17] Omid Bozorg-Haddad, Mohammad Solgi, and Hugo A Loáiciga. *Meta-heuristic and evolutionary algorithms for engineering optimization*. John Wiley & Sons, 2017.
- [BL02] Hyeran Byun and Seong-Whan Lee. Applications of support vector machines for pattern recognition: A survey. In *International workshop on support vector machines*, pages 213–236. Springer, 2002.
- [BMB⁺20] Chaima Boufaied, Claudio Menghi, Domenico Bianculli, Lionel Briand, and Yago Isasi-Parache. Trace-checking signal-based temporal properties: A model-driven approach. In *International Conference on Automated Software Engineering (ASE 2020)*. IEEE, 2020.
- [BNKF98] Wolfgang Banzhaf, Peter Nordin, Robert E Keller, and Frank D Francone. *Genetic programming*. Springer, 1998.
- [BNSB16] Lionel C. Briand, Shiva Nejati, Mehrdad Sabetzadeh, and Domenico Bianculli. Testing the untestable: model testing of complex software-intensive systems. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, pages 789–792, 2016.
- [BOV⁺19] Matthias Bernaerts, Bentley Oakes, Ken Vanherpen, Bjorn Aelvoet, Hans Vangheluwe, and Joachim Denil. Validating industrial requirements with a contract-based approach. In *International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 18–27. IEEE, 2019.
- [BRP16] Marcello M. Bersani, Matteo Rossi, and Pierluigi San Pietro. A tool for deciding the satisfiability of continuous-time metric temporal logic. *Acta Inf.*, 53(2):171–206, 2016. [doi:10.1007/s00236-015-0229-y](https://doi.org/10.1007/s00236-015-0229-y).
- [BT⁺93] Dimitris Bertsimas, John Tsitsiklis, et al. Simulated annealing. *Statistical science*, 8(1):10–15, 1993.
- [C⁺00] International Electrotechnical Commission et al. Functional safety of electrical/electronic/programmable electronic safety related systems. *IEC 61508*, 2000.
- [CAG20] Davide G. Cavezza, Dalal Alrajeh, and András György. Minimal assumptions refinement for realizable specifications. In *International Conference on Formal Methods in Software Engineering (FormaliSE)*. ACM, 2020.
- [CBRZ01] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

- [CGP01] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 2001. URL: <http://books.google.de/books?id=Nmc4wEaLXFEC>.
- [CGP03] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 331–346. Springer, 2003.
- [Cha09] Devendra K. Chaturvedi. *Modeling and Simulation of Systems Using MATLAB and Simulink*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2009.
- [Cha10] D.K. Chaturvedi. *Modeling and Simulation of Systems Using MATLAB and Simulink*. Electrical Engineering. CRC Press, 2010. URL: <https://books.google.lu/books?id=6IW4ngEACAAJ>.
- [Cha17] Devendra K Chaturvedi. *Modeling and simulation of systems using MATLAB and Simulink*. CRC press, 2017.
- [CHS05] D. Coppit and J. M. Haddox-Schatz. On the use of specification-based assertions as test oracles. In *NASA Software Engineering Workshop*, pages 305–314. IEEE, April 2005.
- [CJGK⁺18] Edmund M Clarke Jr, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model checking*. MIT press, 2018.
- [CKMT10] Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. Adaptive random testing: The ART of test case diversity. *J. Syst. Softw.*, 83(1):60–66, 2010.
- [CL08] Christos G. Cassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems, Second Edition*. Springer, 2008. [doi:10.1007/978-0-387-68612-7](https://doi.org/10.1007/978-0-387-68612-7).
- [CLM04] Tsong Yueh Chen, Hing Leung, and I. K. Mak. Adaptive random testing. In *Advances in Computer Science - ASIAN*, pages 320–329. Springer, 2004.
- [CPS16] Yuqi Chen, Christopher M Poskitt, and Jun Sun. Towards learning and verifying invariants of cyber-physical systems by code mutation. In *International Symposium on Formal Methods*, pages 155–163. Springer, 2016.
- [dAH01a] Luca de Alfaro and Thomas A. Henzinger. Interface automata. *ACM SIGSOFT Software Engineering Notes*, 26(5), September 2001. URL: <https://doi.org/10.1145/503271.503226>, [doi:10.1145/503271.503226](https://doi.org/10.1145/503271.503226).
- [dAH01b] Luca de Alfaro and Thomas A. Henzinger. Interface theories for component-based design. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Embedded Software*, pages 148–165. Springer, 2001.
- [Dav87] Lawrence Davis. *Genetic algorithms and simulated annealing*. 1987.
- [DHF14] Adel Dokhanchi, Bardh Hoxha, and Georgios Fainekos. On-line monitoring for temporal logic robustness. In *International Conference on Runtime Verification*, pages 231–246. Springer, 2014.
- [DHKR11] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. Software verification using k-induction. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, pages 351–368, 2011.

- [DLS12] P. Derler, E. A. Lee, and A. Sangiovanni Vincentelli. Modeling cyber-physical systems. *Proceedings of the IEEE*, 100(1):13–28, 2012. [doi:10.1109/JPROC.2011.2160929](https://doi.org/10.1109/JPROC.2011.2160929).
- [DLTT13] Patricia Derler, Edward A Lee, Stavros Tripakis, and Martin Törngren. Cyber-physical system design contracts. In *International Conference on Cyber-Physical Systems*, pages 109–118. ACM, 2013.
- [DM10] Alexandre Donzé and Oded Maler. Robust satisfaction of temporal logic over real-valued signals. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 92–106. Springer, 2010.
- [DMB08a] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS’08/ETAPS’08*, pages 337–340, 2008.
- [DMB08b] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
- [Don10] Alexandre Donzé. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *Computer Aided Verification*, pages 167–170. Springer, 2010.
- [DR96] L. K. Dillon and Y. S. Ramakrishna. Generating oracles from your favorite temporal logic specifications. In *Symposium on Foundations of Software Engineering, SIGSOFT*. ACM, 1996.
- [dWOJ⁺07] Eckert Claudia M de Weck Olivier, Clarkson P John, et al. A classification of uncertainty for early product and system design. *Guidelines for a Decision Support Method Adapted to NPD Processes*, pages 159–160, 2007.
- [DY94] Laura K Dillon and Qing Yu. Specification and testing of temporal properties of concurrent system designs. *University of California at Santa Barbara*, 1994.
- [EAB⁺20] Gidon Ernst, Paolo Arcaini, Ismail Bennani, Alexandre Donze, Georgios Fainekos, Goran Frehse, Logan Mathesen, Claudio Menghi, Giulia Pedrielli, Marc Pouzet, Shakiba Yaghoubi, Yoriyuki Yamagata, and Zhenya Zhang. ARCH-COMP 2020 category report: Falsification. In *International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH20)*, volume 74 of *EPiC Series in Computing*, pages 140–152. EasyChair, 2020.
- [EAD⁺19] Gidon Ernst, Paolo Arcaini, Alexandre Donze, Georgios Fainekos, Logan Mathesen, Giulia Pedrielli, Shakiba Yaghoubi, Yoriyuki Yamagata, and Zhenya Zhang. Arch-comp 2019 category report: Falsification. *EPiC Series in Computing*, 61:129–140, 2019.
- [EPG⁺07a] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35 – 45, 2007. [doi:https://doi.org/10.1016/j.scico.2007.01.015](https://doi.org/10.1016/j.scico.2007.01.015).

- [EPG⁺07b] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Science of computer programming*, 69(1–3):35–45, 2007. doi: [10.1016/j.scico.2007.01.015](https://doi.org/10.1016/j.scico.2007.01.015).
- [Epi20] Epicurus material, 2020. URL: <https://github.com/SNTSVV/EPIcuRus>.
- [ER14] Sebastian Elbaum and David S Rosenblum. Known unknowns: testing in the presence of uncertainty. In *International Symposium on Foundations of Software Engineering*, pages 833–836. ACM, 2014.
- [ESA20a] ESA. Building and testing spacecraft, 2020. URL: https://www.esa.int/Science_Exploration/Space_Science/Building_and_testing_spacecraft.
- [ESA20b] The European Space Agency (ESA), 2020. URL: <https://www.esa.int/>.
- [exa20] exactEarth, 2020. URL: <https://www.exactearth.com/>.
- [FGD⁺11] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. SpaceEx: Scalable Verification of Hybrid Systems. In *Computer Aided Verification (CAV)*. Springer, 2011.
- [FW16] Eibe Frank, Mark A Hall, and Ian H Witten. *The WEKA workbench*. Morgan Kaufmann, 2016.
- [fit20] fitctree, 2020. URL: <https://nl.mathworks.com/help/stats/fitctree.html>.
- [FL01] Cormac Flanagan and K. Rustan M. Leino. Houdini, an Annotation Assistant for ESC/Java. In *International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity (FME)*, page 500–517. Springer-Verlag, 2001.
- [flo20] floor, 2020. URL: <https://www.mathworks.com/help/matlab/ref/floor.html>.
- [FP09] Georgios E Fainekos and George J Pappas. Robustness of temporal logic specifications for continuous-time signals. *Theoretical Computer Science*, 410(42):4262–4291, 2009.
- [FR07] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *Future of Software Engineering (FOSE '07)*, pages 37–54, 2007. doi: [10.1109/FOSE.2007.14](https://doi.org/10.1109/FOSE.2007.14).
- [GCPT19] Divya Gopinath, Hayes Converse, Corina S. Păsăreanu, and Ankur Taly. Property inference for deep neural networks. In *International Conference on Automated Software Engineering (ASE)*, page 797–809. IEEE, 2019.
- [GFH16] Kerianne H. Gross, Aaron W. Ficarek, and Jonathan A. Hoffman. Incremental formal methods based design approach demonstrated on a coupled tanks control system. In *17th IEEE International Symposium on High Assurance Systems Engineering, HASE 2016, Orlando, FL, USA, January 7-9, 2016*, pages 181–188, 2016.
- [GK10] Farid Golnaraghi and BC Kuo. Automatic control systems. *Complex Variables*, 2:1–1, 2010.

- [GLMN14] Pranav Garg, Christof Loding, P. Madhusudan, and Daniel Neider. ICE: A robust framework for learning invariants. In *Computer Aided Verification (CAV)*. Springer, 2014.
- [GMN⁺20] Khouloud Gaaloul, Claudio Menghi, Shiva Nejati, Lionel C. Briand, and David Wolfe. Mining assumptions for software components using machine learning. In *Foundations of Software Engineering (ESEC/FSE)*. ACM, 2020.
- [GMN⁺55] K. Gaaloul, C. Menghi, S. Nejati, L. Briand, and Y. Isasi Parache. Combining genetic programming and model checking to generate environment assumptions. *IEEE Transactions on Software Engineering*, (01):1–1, aug 5555. doi:10.1109/TSE.2021.3101818.
- [GMNB20] Khouloud Gaaloul, Claudio Menghi, Shiva Nejati, and Lionel Briand. Epicurus. Zenodo, Jun 2020. doi:10.5281/zenodo.3872902.
- [GNMR16] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. Learning invariants using decision trees and implication counterexamples. *SIGPLAN Not.*, 51(1):499–512, 2016. doi:10.1145/2914770.2837664.
- [GPB02] Dimitra Giannakopoulou, Corina S Pasareanu, and Howard Barringer. Assumption generation for software component verification. In *International Conference on Automated Software Engineering*, pages 3–12. IEEE, 2002.
- [GPB08] Dimitra Giannakopoulou, Corina S. Pasareanu, and Colin Blundell. Assume-guarantee testing for software components. *IET Software*, 2(6):547–562, 2008. doi:10.1049/iet-sen:20080012.
- [GPC04] Dimitra Giannakopoulou, Corina S Pasareanu, and Jamieson M Cobleigh. Assume-guarantee verification of source code with design-level assumptions. In *International Conference on Software Engineering*, pages 211–220. IEEE, 2004.
- [GPM20] Matlab GP toolbox., 2020. URL: <https://it.mathworks.com/matlabcentral/fileexchange/47197-genetic-programming-matlab-toolbox>.
- [GPS⁺17] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.
- [Ham08] Gregoire Hamon. Simulink Design Verifier - Applying Automated Formal Methods to Simulink and Stateflow. In *AFM 2008*. Citeseer, 2008.
- [HDE⁺08] Grégoire Hamon, Bruno Dutertre, Levent Erkok, John Matthews, Daniel Sheridan, David Cok, John Rushby, Peter Bokor, Sandeep Shukla, Andras Pataricza, et al. Simulink design verifier-applying automated formal methods to simulink and stateflow. In *Third Workshop on Automated Formal Methods*, 2008.
- [Hen09] Harry Henderson. *Encyclopedia of computer science and technology*. Infobase Publishing, 2009.
- [HKPV95] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What’s decidable about hybrid automata? In *Annual ACM Symposium on Theory of Computing (STOC)*. ACM, 1995.

- [HKPV98] Thomas A Henzinger, Peter W Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? *Journal of computer and system sciences*, 57(1):94–124, 1998.
- [HMF15] Bardh Hoxha, Nikolaos Mavridis, and Georgios Fainekos. Vispec: A graphical tool for elicitation of mtl requirements. In *Intelligent Robots and Systems (IROS)*, pages 3486–3492. IEEE, 2015.
- [HMSY13] Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. A comprehensive survey of trends in oracles for software testing. *University of Sheffield, Department of Computer Science, Tech. Rep. CS-13-01*, 2013.
- [HMZ12] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, 45(1):11:1–11:61, 2012. [doi:10.1145/2379776.2379787](https://doi.org/10.1145/2379776.2379787).
- [HQR98] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. You assume, we guarantee: Methodology and case studies. In *Computer Aided Verification*, pages 440–451. Springer, 1998.
- [HSX⁺19] Rubing Huang, Weifeng Sun, Yinyin Xu, Haibo Chen, Dave Towey, and Xin Xia. A survey on adaptive random testing. *IEEE Transactions on Software Engineering*, 2019.
- [JBGN16] Stefan Jakšić, Ezio Bartocci, Radu Grosu, and Dejan Ničković. Quantitative monitoring of stl with edit distance. In *International Conference on Runtime Verification*, pages 201–218. Springer, 2016.
- [JCL11] Jeff C. Jensen, Danica H. Chang, and Edward A. Lee. A model-based design methodology for cyber-physical systems. In *2011 7th International Wireless Communications and Mobile Computing Conference*, pages 1666–1671, 2011. [doi:10.1109/IWCMC.2011.5982785](https://doi.org/10.1109/IWCMC.2011.5982785).
- [JDDS15] Xiaoqing Jin, Alexandre Donzé, Jyotirmoy V. Deshmukh, and Sanjit A. Seshia. Mining requirements from closed-loop control models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(11):1704–1717, 2015. [doi:10.1109/TCAD.2015.2421907](https://doi.org/10.1109/TCAD.2015.2421907).
- [JDVDAS13] Xiaoqing Jin, Alexandre Donzé, Jyotirmoy V. Deshmukh, and Sanjit A. Seshia. Mining requirements from closed-loop control models. In *International conference on Hybrid systems: computation and control, HSCC*. ACM, 2013.
- [KBD⁺17] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. *CoRR*, abs/1702.01135, 2017. URL: <http://arxiv.org/abs/1702.01135>.
- [KBD⁺20] Maureen Keegan, Victor A Braberman, Nicolas D’Ippolito, Nir Piterman, and Sebastian Uchitel. Control and discovery of environment behaviour. *IEEE Transactions on Software Engineering*, 2020.
- [KDJ⁺16] J. Kapinski, J. V. Deshmukh, X. Jin, H. Ito, and K. Butts. 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):45–64, 2016.
- [KHSZ20] Alexander Kampmann, Nikolas Havrikov, Ezekiel O. Soremekun, and Andreas Zeller. When does my program do this? learning circumstances of software behavior. In *Foundations of Software Engineering (ESEC/FSE)*. ACM, 2020.
- [Kit09] Emanuel Kitzelmann. Inductive programming: A survey of program synthesis techniques. In *International workshop on approaches and applications of inductive programming*, pages 50–73. Springer, 2009.
- [KK92] John R Koza and John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.
- [KK00] John R. Koza and Martin A. Keane. Automatic creation of human-competitive programs and controllers by means of genetic programming, 2000.
- [Koy90a] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-time systems*, 2(4):255–299, 1990.
- [Koy90b] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Syst.*, 2(4):255–299, October 1990. URL: <https://doi.org/10.1007/BF01995674>, doi:10.1007/BF01995674.
- [KZP07] Sotiris B Kotsiantis, I Zaharakis, and P Pintelas. Supervised machine learning: A review of classification techniques. *Emerging artificial intelligence applications in computer engineering*, 160(1):3–24, 2007.
- [LDS11] Wenchao Li, Lili Dworkin, and Sanjit A. Seshia. Mining assumptions for synthesis. In *International Conference on Formal Methods and Models*, pages 43–50. IEEE, 2011.
- [Lee08] Edward A. Lee. Cyber physical systems: Design challenges. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369, 2008. doi:10.1109/ISORC.2008.25.
- [LGA96] Pascale Le Gall and Agnès Arnould. Formal specifications and test: Correctness and oracle. In Magne Haverlaen, Olaf Owe, and Ole-Johan Dahl, editors, *Recent Trends in Data Type Specification*. Springer, 1996.
- [LH01] Jin-Cherng Lin and Ian Ho. Generating timed test cases with oracles for real-time software. *Advances in Engineering Software*, 32(9):705 – 715, 2001. URL: <http://www.sciencedirect.com/science/article/pii/S0965997801000217>, doi:[https://doi.org/10.1016/S0965-9978\(01\)00021-7](https://doi.org/10.1016/S0965-9978(01)00021-7).
- [loc20] Lockheed Martin, 2020. URL: <https://www.lockheedmartin.com/en-us/index.html>.
- [Lon11] Michael A. Lones. Sean luke: essentials of metaheuristics. *Genet. Program. Evolvable Mach.*, 12(3):333–334, 2011.
- [LT88] Kim G Larsen and Bent Thomsen. A modal process logic. In *Logic in Computer Science*, pages 203–210. IEEE, 1988.

- [Luk13] Sean Luke. *Essentials of Metaheuristics*. Lulu, second edition, 2013.
- [Lux19] Luxspace, 2019. URL: <https://luxspace.lu/>.
- [MAES19] Urtzi Markiegi, Aitor Arrieta, Leire Etxeberria, and Goiuria Sagardui. Test case selection using structural coverage in software product lines for time-budget constrained scenarios. In *SIGAPP Symposium on Applied Computin (SAC)*, page 2362–2371. ACM, 2019.
- [Mar97] Anu Maria. Introduction to modeling and simulation. In *Proceedings of the 29th conference on Winter simulation*, pages 7–13, 1997.
- [MAS05] János Madár, János Abonyi, and Ferenc Szeifert. Genetic programming for the identification of nonlinear input- output models. *Industrial & engineering chemistry research*, 44(9):3178–3186, 2005.
- [mat] Mathematica. <https://www.wolfram.com/mathematica/>. Accessed: 2019-04-26.
- [mat19] Mathworks. <https://mathworks.com>, 02 2019.
- [MBC10] Mauro Mazzolini, Alessandro Brusaferrri, and Emanuele Carpanzano. Model-checking based verification approach for advanced industrial automation solutions. In *2010 IEEE 15th Conference on Emerging Technologies & Factory Automation (ETFA 2010)*, pages 1–8. IEEE, 2010.
- [MBG⁺20a] A. Mavridou, H. Bourbough, D. Giannakopoulou, T. Pressburger, M. Hejase, P. L. Garoche, and J. Schumann. The Ten Lockheed Martin Cyber-Physical Challenges: Formalized, Analyzed, and Explained. In *International Requirements Engineering Conference (RE)*, pages 300–310. IEEE, 2020.
- [MBG⁺20b] Anastasia Mavridou, Hamza Bourbough, Pierre-Loïc Garoche, Dimitra Giannakopoulou, Thomas Pressburger, and Johann Schumann. Bridging the gap between requirements and simulink model analysis. In *REFSQ Workshops*, 2020.
- [MBG⁺20c] Anastasia Mavridou, Hamza Bourbough, Pierre-Loïc Garoche, Dimitra Giannakopoulou, Tom Pressburger, and Johann Schumann. Bridging the gap between requirements and simulink model analysis. In *Requirements Engineering: Foundation for Software Quality (REFSQ), Companion Proceedings*. Springer, 2020.
- [MBRP20] Claudio Menghi, Marcello M. Bersani, Matteo Rossi, and Pierluigi San Pietro. Model checking mitl formulae on timed automata: a logic-based approach. *Transactions on Computational Logic*, 2020.
- [McD09] John H McDonald. *Handbook of biological statistics*, volume 2. sparky house publishing Baltimore, MD, 2009.
- [McM04] Phil McMinn. Search-based software test data generation: A survey: Research articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, June 2004.
- [mex20] MEX function, 2020. URL: <https://it.mathworks.com/help/matlab/call-mex-file-functions.html>.
- [MH93] Melanie Mitchell and John H Holland. When will a genetic algorithm outperform hill-climbing? 1993.

- [Mil09] Steven P Miller. Bridging the gap between model-based development and model checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 443–453. Springer, 2009.
- [MMM95] Dino Mandrioli, Sandro Morasca, and Angelo Morzenti. Generating test cases for real-time systems from logic specifications. *ACM Trans. Comput. Syst.*, 13(4):365–398, November 1995. URL: <http://doi.acm.org/10.1145/210223.210226>, doi: [10.1145/210223.210226](https://doi.org/10.1145/210223.210226).
- [MMS96] Sandro Morasca, Angelo Morzenti, and Pieluigi SanPietro. Generating functional test cases in-the-large for time-critical systems from logic-based specifications. In *1996 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '96*, pages 39–52, New York, NY, USA, 1996. ACM. URL: <http://doi.acm.org/10.1145/229000.226300>, doi: [10.1145/229000.226300](https://doi.org/10.1145/229000.226300).
- [MN04] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166. Springer, 2004.
- [MN13] Oded Maler and Dejan NižKović. Monitoring properties of analog and mixed-signal circuits. *Int. J. Softw. Tools Technol. Transf.*, 15(3):247–268, June 2013. URL: <https://doi.org/10.1007/s10009-012-0247-9>, doi: [10.1007/s10009-012-0247-9](https://doi.org/10.1007/s10009-012-0247-9).
- [MNB⁺13] Reza Matinnejad, Shiva Nejati, Lionel Briand, Thomas Bruckmann, and Claude Poull. Automated model-in-the-loop testing of continuous controllers using search. In *International Symposium on Search Based Software Engineering*, pages 141–157. Springer, 2013.
- [MNB⁺15a] Reza Matinnejad, Shiva Nejati, Lionel Briand, Thomas Bruckmann, and Claude Poull. Search-based automated testing of continuous controllers: Framework, tool support, and case studies. *Information and Software Technology*, 57:705–722, 2015.
- [MNB⁺15b] Reza Matinnejad, Shiva Nejati, Lionel C. Briand, Thomas Bruckmann, and Claude Poull. Search-based automated testing of continuous controllers: Framework, tool support, and case studies. *Information & Software Technology*, 57:705–722, 2015.
- [MNB17a] Reza Matinnejad, Shiva Nejati, and Lionel C. Briand. Automated testing of hybrid simulink/stateflow controllers: industrial case studies. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*, pages 938–943, 2017.
- [MNB17b] Reza Matinnejad, Shiva Nejati, and Lionel C Briand. Automated testing of hybrid simulink/stateflow controllers: industrial case studies. In *Foundations of Software Engineering*, pages 938–943. ACM, 2017.
- [MNBB14] Reza Matinnejad, Shiva Nejati, Lionel C. Briand, and Thomas Bruckmann. Mil testing of highly configurable continuous controllers: scalable search using surrogate models. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 163–174, 2014.
- [MNBB16a] Reza Matinnejad, Shiva Nejati, Lionel C. Briand, and Thomas Bruckmann. Automated test suite generation for time-continuous simulink models. In *International Conference on Software Engineering, ICSE*. ACM, 2016.

- [MNBB16b] Reza Matinnejad, Shiva Nejati, Lionel C Briand, and Thomas Bruckmann. Automated test suite generation for time-continuous simulink models. In *proceedings of the 38th International Conference on Software Engineering*, pages 595–606, 2016.
- [MNBB16c] Reza Matinnejad, Shiva Nejati, Lionel C Briand, and Thomas Bruckmann. Simcotest: A test suite generation tool for simulink/stateflow controllers. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 585–588, 2016.
- [MNBB18] Reza Matinnejad, Shiva Nejati, Lionel Briand, and Thomas Bruckmann. Test generation and test prioritization for simulink models with dynamic behavior. *IEEE Transactions on Software Engineering*, 2018.
- [MNBP20] Claudio Menghi, Shiva Nejati, Lionel C. Briand, and Yago Isasi Parache. Approximation-refinement testing of compute-intensive cyber-physical models: An approach based on system identification. In *International Conference on Software Engineering (ICSE)*. ACM, 2020.
- [MNGB19] Claudio Menghi, Shiva Nejati, Khoulood Gaaloul, and Lionel C. Briand. Generating automated and online test oracles for simulink models with continuous and uncertain behaviors. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE*. ACM, 2019.
- [Mol19a] Christoph Molnar. *Interpretable Machine Learning*. 2019. <https://christophm.github.io/interpretable-ml-book/>.
- [Mol19b] Christoph Molnar. *Interpretable machine learning*. Lulu. com, 2019.
- [Mon95] David J Montana. Strongly typed genetic programming. *Evolutionary computation*, 3(2):199–230, 1995.
- [MRS19] Shahar Maoz, Jan Oliver Ringert, and Rafi Shalom. Symbolic repairs for GR(1) specifications. In *International Conference on Software Engineering (ICSE)*, pages 1016–1026. IEEE, 2019.
- [MSCC18] Claudio Menghi, Paola Spoletini, Marsha Checkik, and Ghezzi Carlo. Supporting verification-driven incremental distributed design of components. In *Fundamental Approaches to Software Engineering (FASE)*. Springer, 2018.
- [MSCG19] Claudio Menghi, Paola Spoletini, Marsha Chechik, and Carlo Ghezzi. A verification-driven framework for iterative design of controllers. *Formal Aspects of Computing*, 31(5):459–502, 2019.
- [Mur98] Sreerama K Murthy. Automatic construction of decision trees from data: A multi-disciplinary survey. *Data mining and knowledge discovery*, 2(4):345–389, 1998.
- [MVBB21] Claudio Menghi, Enrico Viganò, Domenico Bianculli, and Lionel C. Briand. Trace-Checking CPS Properties: Bridging the Cyber-Physical Gap. In *International Conference on Software Engineering (ICSE)*. ACM, 2021.
- [ND15] Paulo A Nardi and Eduardo F Damasceno. A survey on test oracles. 2015.

- [NDB13a] P. A. Nardi, M. E. Delamaro, and L. Baresi. Specifying automated oracles for simulink models. In *International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 330–333. IEEE, 2013.
- [NDB13b] Paulo A Nardi, Marcio E Delamaro, and Luciano Baresi. Specifying automated oracles for simulink models. In *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 330–333. IEEE, 2013.
- [Nej20] Shiva Nejati. Search-based software testing for formal software verification – and vice versa. In Aldeida Aleti and Annibale Panichella, editors, *Search-Based Software Engineering*, pages 3–6, Cham, 2020. Springer International Publishing.
- [New74] Isaac Newton. Methodus fluxionum et seriarum infinitarum. *Opuscula mathematica, philosophica et philologica*, 1, 1774.
- [NFIS14] Pierluigi Nuzzo, John B. Finn, Antonio Iannopolo, and Alberto L. Sangiovanni-Vincentelli. 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, 2014.
- [NGM⁺19] Shiva Nejati, Khoulood Gaaloul, Claudio Menghi, Lionel C Briand, Stephen Foster, and David Wolfe. Evaluating model testing and model checking for finding requirements violations in simulink models. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE*, pages 1015–1025. ACM, 2019.
- [Nis04] N. S. Nise. *Control Systems Engineering*. John-Wiley Sons, 4th edition, 2004.
- [NXO⁺14] Pierluigi Nuzzo, Huan Xu, Necmiye Ozay, John B. Finn, Alberto L. Sangiovanni-Vincentelli, Richard M. Murray, Alexandre Donzé, and Sanjit A. Seshia. A contract-based methodology for aircraft electric power system design. *IEEE Access*, 2:1–25, 2014. [doi:10.1109/ACCESS.2013.2295764](https://doi.org/10.1109/ACCESS.2013.2295764).
- [OKN14] Rafael AP Oliveira, Upulee Kanewala, and Paulo A Nardi. Automated test oracles: State of the art, taxonomies, and trends. In *Advances in computers*, volume 95, pages 113–199. Elsevier, 2014.
- [PBL98] Riccardo Poli and William B. Langdon. Schema theory for genetic programming with one-point crossover and point mutation. In *Evolutionary Computation*, volume 6, pages 231–252. 1998.
- [PHPS03] J Philipps, G Hahn, A Pretschner, and T Stauner. Tests for mixed discrete-continuous reactive systems. In *Proc. 14th IEEE Int. Workshop Rapid Syst. Prototyping*, pages 78–84, 2003.
- [PKB⁺20] Nir Piterman, Maureen Keegan, Victor Braberman, Nicolas D’Ippolito, and Sebastian Uchitel. Control and discovery of reactive system environments. In *University of Leicester*, 2020.
- [PL98] Riccardo Poli and William B Langdon. Genetic programming with one-point crossover. In *Soft Computing in Engineering Design and Manufacturing*, pages 180–189. Springer, 1998.

- [PLMK08] Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. *A field guide to genetic programming*. Lulu. com, 2008.
- [Pnu77] Amir Pnueli. The temporal logic of programs. In *Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. IEEE, 1977.
- [Pro] Prover Technology. Prover Plug-In Software. <http://www.prover.com>. [Online; accessed 17-Aug-2015].
- [PW18] Ingo Pill and Franz Wotawa. Automated generation of (f)ltl oracles for testing and debugging. *Journal of Systems and Software*, 139:124–141, 2018.
- [qra19] QRA corp, 2019. URL: <https://qracorp.com/>.
- [Qui86] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.
- [Qui87] J. Ross Quinlan. Simplifying decision trees. *International journal of man-machine studies*, 27(3):221–234, 1987.
- [RAO92] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O’Malley. Specification-based test oracles for reactive systems. In *International Conference on Software Engineering, ICSE*. ACM, 1992.
- [REA19] reactive-systems. <https://www.reactive-systems.com>, 02 2019.
- [RM05] Lior Rokach and Oded Maimon. Decision trees. In *Data mining and knowledge discovery handbook*, pages 165–192. Springer, 2005.
- [Rob19] Robust control toolbox. <https://nl.mathworks.com/products/robust.html>, 02 2019.
- [Rog72] David Rogers. Random search and insect population models. *The Journal of Animal Ecology*, pages 369–383, 1972.
- [RS11] Pritam Roy and Natarajan Shankar. Simcheck: a contract type system for simulink. *Innovations in Systems and Software Engineering*, 7(2):73–83, 2011.
- [RU12] Nicholas Rescher and Alasdair Urquhart. *Temporal logic*, volume 3. Springer Science & Business Media, 2012.
- [Rut89] Rob A Rutenbar. Simulated annealing algorithms: An overview. *IEEE Circuits and Devices magazine*, 5(1):19–26, 1989.
- [RW87] Peter J Ramadge and W Murray Wonham. Supervisory control of a class of discrete event processes. *SIAM journal on control and optimization*, 25(1):206–230, 1987.
- [RWW89] Peter JG Ramadge and Murray Wonham W. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.
- [SA03] Sara Silva and Jonas Almeida. Gplab-a genetic programming toolbox for Matlab. In *Proceedings of the Nordic MATLAB conference*, pages 273–278. Citeseer, 2003.
- [SBB⁺18] Joshua Schneider, David Basin, Frederik Brix, Srđan Krstić, and Dmitriy Traytel. Scalable online first-order monitoring. In Christian Colombo and Martin Leucker, editors, *Runtime Verification*, pages 353–371, Cham, 2018. Springer.

- [SBLR07] Manoranjan Satpathy, Michael Butler, Michael Leuschel, and S. Ramesh. Automatic testing from formal specifications. In Yuri Gurevich and Bertrand Meyer, editors, *Tests and Proofs*, pages 95–113, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
- [SC93] P. A. Stocks and D. A. Carrington. Test templates: A specification-based testing framework. In *International Conference on Software Engineering, ICSE*. IEEE, 1993.
- [SCN⁺20] Seung Yeob Shin, Karim Chaouch, Shiva Nejati, Mehrdad Sabetzadeh, Lionel C Briand, and Frank Zimmer. Uncertainty-aware specification and analysis for hardware-in-the-loop testing of cyber-physical systems. *Journal of Systems and Software*, 171:110813, 2020.
- [SDV19] Simulink Design Verifier. <https://nl.mathworks.com/products/sldesignverifier.html>, 02 2019.
- [Sea15] Dominic P Searson. GPTIPS 2: an open-source software platform for symbolic data mining. In *Handbook of genetic programming applications*, pages 551–573. Springer, 2015.
- [sfu20] S-Function, 2020. URL: <https://www.mathworks.com/help/simulink/sfg/what-is-an-s-function.html>.
- [Sig19] Signal To Noise Ratio. https://en.wikipedia.org/wiki/Signal-to-noise_ratio, 02 2019.
- [Sim19] Simulink solvers. <https://nl.mathworks.com/help/simulink/ug/types-of-solvers.html>, 02 2019.
- [Sim20] Simulink, 2020. URL: <https://nl.mathworks.com/products/simulink.html>.
- [sim21] simulation and model-based design. <https://www.mathworks.com/products/simulink.html>, 06 2021.
- [sir19] Sirius. <https://www.eclipse.org/sirius/>, 02 2019.
- [SL91] S Rasoul Safavian and David Landgrebe. A survey of decision tree classifier methodology. *IEEE transactions on systems, man, and cybernetics*, 21(3):660–674, 1991.
- [SL02] J. Srinivasan and N. Leveson. Automated testing from specifications. In *Digital Avionics Systems Conference*, pages 6A2–6A2. IEEE, 2002.
- [SMP⁺18] Alexander Schaap, Gordon Marks, Vera Pantelic, Mark Lawford, Gehan Selim, Alan Wassyng, and Lucian Patcas. Documenting simulink designs of embedded systems. In *International Conference on Model Driven Engineering Languages and Systems (MODELS): Companion Proceedings*, page 47–51. ACM, 2018.
- [SNBB18] Simone Silveti, Laura Nenzi, Ezio Bartocci, and Luca Bortolussi. Signal convolution logic. *arXiv preprint arXiv:1806.00238*, 2018.
- [SNWS09] X. Sun, P. Nuzzo, C. Wu, and A. Sangiovanni-Vincentelli. Contract-based system-level composition of analog circuits. In *Design Automation Conference*. ACM, 2009.
- [Soc19] Socrates material, 2019. URL: <https://github.com/SNTSVV/SOCraTES>.

- [SSC⁺04] Norman Scaife, Christos Sofronis, Paul Caspi, Stavros Tripakis, and Florence Maranchi. Defining and translating a "safe" subset of simulink/stateflow into lustre. In *Proceedings of the 4th ACM international conference on Embedded software*, pages 259–268, 2004.
- [STB⁺06] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2006.
- [sub20] Subsystems, 2020. URL: <https://it.mathworks.com/help/simulink/subsystems.html>.
- [SVDP12] Alberto Sangiovanni-Vincentelli, Werner Damm, and Roberto Passerone. Taming dr. frankenstein: Contract-based design for cyber-physical systems. *European journal of control*, 18(3):217–238, 2012.
- [SWYS11] Jianhua Shi, Jiafu Wan, Hehua Yan, and Hui Suo. A survey of cyber-physical systems. In *2011 International Conference on Wireless Communications and Signal Processing (WCSP)*, pages 1–6, 2011. [doi:10.1109/WCSP.2011.6096958](https://doi.org/10.1109/WCSP.2011.6096958).
- [Tal09] El-Ghazali Talbi. *Metaheuristics: from design to implementation*, volume 74. John Wiley & Sons, 2009.
- [TFP⁺19] Cumhuri Erkan Tuncali, Georgios Fainekos, Danil Prokhorov, Hisahiro Ito, and James Kapinski. Requirements-driven test generation for autonomous vehicles with machine learning components. *IEEE Transactions on Intelligent Vehicles*, 5(2):265–280, 2019.
- [Tiw12] Ashish Tiwari. Hybridsal relational abstracter. In *Computer Aided Verification (CAV)*. Springer, 2012.
- [VBCG14] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos. Management of an Academic HPC Cluster: The UL Experience. In *Proc. of the 2014 Intl. Conf. on High Performance Computing & Simulation (HPCS 2014)*, pages 959–967, Bologna, Italy, July 2014. IEEE.
- [vir20] Virtual vector, 2020. URL: <https://it.mathworks.com/help/simulink/ug/signal-types.html>.
- [VLA87] Peter JM Van Laarhoven and Emile HL Aarts. Simulated annealing. In *Simulated annealing: Theory and applications*, pages 7–15. Springer, 1987.
- [Wai09] Gabriel A. Wainer. *Discrete-Event Modeling and Simulation: A Practitioner's Approach*. CRC Press, Inc., USA, 1st edition, 2009.
- [WFHP16] Ian H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. *Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2016.
- [Wie98] B. Wie. *Space Vehicle Dynamics and Control*. AIAA education series. American Institute of Aeronautics and Astronautics, 1998.

- [Win09] Andreas Windisch. Search-based testing of complex simulink models containing stateflow diagrams. In *2009 31st International Conference on Software Engineering-Companion Volume*, pages 395–398. IEEE, 2009.
- [Win10] Andreas Windisch. Search-based test data generation from stateflow statecharts. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 1349–1356, 2010.
- [WKLS18] Kosuke Watanabe, Eunsuk Kang, Chung-Wei Lin, and Shinichi Shiraishi. Runtime monitoring for safety of intelligent vehicles. In *Annual Design Automation Conference, DAC*. ACM, 2018.
- [WRDM96] Darrell Whitley, Soraya Rana, John Dzubera, and Keith E Mathias. Evaluating evolutionary algorithms. *Artificial intelligence*, 85(1-2):245–276, 1996.
- [WSB⁺09] Jon Whittle, Pete Sawyer, Nelly Bencomo, Betty HC Cheng, and Jean-Michel Bruel. Relax: Incorporating uncertainty into the specification of self-adaptive systems. In *Requirements Engineering Conference*, pages 79–88. IEEE, 2009.
- [Xte19] Xtext. <https://www.eclipse.org/Xtext/>, 02 2019.
- [Yan11] Xin-She Yang. Metaheuristic optimization. *Scholarpedia*, 6(8):11472, 2011.
- [ZC06] Ji Zhang and Betty HC Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the 28th international conference on Software engineering*, pages 371–380, 2006.
- [ZC08] Yuan Zhan and John A Clark. A search-based framework for automatic testing of matlab/simulink models. *Journal of Systems and Software*, 81(2):262–285, 2008.
- [ZJKK17] Xi Zheng, Christine Julien, Miryung Kim, and Sarfraz Khurshid. Perceptions on the state of the art in verification and validation in cyber-physical systems. *Systems Journal*, 11(4):2614–2627, 2017.
- [ZSM12] Justyna Zander, Ina Schieferdecker, and Pieter J Mosterman. *Model-based testing for embedded systems*. CRC Press, 2012.