Test Case Generation for Drivability Requirements of an Automotive Cruise Controller: An Experience with an Industrial Simulator

Federico Formica
McMaster University
Hamilton, Canada

Nicholas Petrunti
McMaster University
Hamilton, Canada

Lucas Bruck
McMaster University
Hamilton, Canada

Vera Pantelic
McMaster University
Hamilton, Canada

Mark Lawford
McMaster University
Hamilton, Canada

Claudio Menghi
University of Bergamo
Bergamo, Italy
McMaster University
Hamilton, Canada

ABSTRACT

Automotive software development requires engineers to test their systems to detect violations of both functional and drivability requirements. Functional requirements define the functionality of the automotive software. Drivability requirements refer to the driver's perception of the interactions with the vehicle; for example, they typically require limiting the acceleration and jerk perceived by the driver within given thresholds. While functional requirements are extensively considered by the research literature, drivability requirements garner less attention.

This industrial paper describes our experience assessing the usefulness of an automated search-based software testing (SBST) framework in generating failure-revealing test cases for functional and drivability requirements. Our experience concerns the VI-CarRealTime simulator, an industrial virtual modeling and simulation environment widely used in the automotive domain. We designed a Cruise Control system in Simulink® for a four-wheel vehicle, in an iterative fashion, by producing 21 model versions. We used the SBST framework for each version of the model to search for failure-revealing test cases revealing requirement violations. Our results show that the SBST framework successfully identified a failure-revealing test case for 66.7% of our model versions, requiring, on average, 245.9s and 3.8 iterations. We present lessons learned, reflect on the generality of our results, and discuss how our results improve the state of practice.

1 INTRODUCTION

Search-based software testing (SBST) is a technique used to automatically generate test cases that show the violation of the system requirements [39]. SBST frameworks support software development in many domains, such as real-time, concurrent, distributed, embedded, and safety-critical [8]. SBST is also extensively used in the context of cyber-physical systems (CPS), including automotive applications (e.g., [14, 43, 51, 67, 68]). Despite their extensive adoption, the usefulness of SBST depends on the assumptions and peculiarities of the different domains since the types of requirements under consideration usually depend on the application domains, e.g., slow response time is a fault relevant for real-time systems [8], but not necessarily for others.

This work focuses on the automotive domain and specifically on *Cruise Controller* (CC) development. CC is a software component that regulates the vehicle's speed. Their global market was valued at USD 34.7 billion in 2022 and is expected to grow to USD 65 billion by 2032 [53], driven by the increasing demand for safety and comfort, the technology advancements, and the rise of autonomous cars [45]. CC needs to satisfy functional requirements while ensuring the comfort of the driver [9, 20]. *Drivability requirements* specify properties related to the comfort of the driver and pilot the CC development [4, 9, 11, 18, 52, 61, 66].

Testing is one of the techniques used to search for violations of the CC requirements [17, 19, 37, 48, 57]. The assessment of the usefulness of testing techniques in practice via empirical studies is fundamental for the academic and industrial communities, which need experimental evaluation results to drive their (business) decisions and understand the advantages and limitations of SBST support for CC development. However, for CC development, most of the studies publicly available in the research literature (e.g., [26, 37, 46, 47, 74, 75]) focus on functional requirements and do not empirically assess the SBST frameworks by considering the versions of the model produced during the incremental and iterative development of CC. This paper addresses this limitation by empirically assessing the usefulness of SBST in detecting failure-revealing test cases for drivability requirements during the end-to-end development of a CC for an industrial simulator.

Our *case study* is the VI-CarRealTime (VICRT) Simulator [77], an industrial real-time simulation environment widely used by automotive companies (e.g., Brembo [1]). VICRT enables engineers to develop control software for their vehicles. It integrates with well-known development tools for cyber-physical system design, such as Matlab/Simulink® [5], and supports both software-in-the-loop (SIL) and hardware-in-the-loop (HIL) simulations. We considered Simulink® for developing the CC since it is a widely used framework for developing control systems in the industry [21, 44]. For the assessment of SBST, we use Simulink® Test™ [6] for test case definition, since it is a standard tool for test case specification in Simulink®. Simulink® Test™ enables engineers to specify test

cases using Simulink® Test Sequence [71] and Test Assessment [70] blocks, also called for short Test Blocks in this work. *Test Sequences* specify test inputs, while *Test Assessments* specify the procedure to check the system requirements. We selected HECATE [35] as our SBST framework since it generates test cases in collaboration with Test Blocks.

We iteratively and incrementally developed a complex CC for our industrial simulator. The development activity required approximately 300 hours spread over eight months, during which we produced 21 versions of the CC referring to seven major versions. We extensively used the SBST framework during the CC development by running SIL and HIL experiments, and assessed the usefulness of SBST by measuring its effectiveness, i.e., how helpful SBST is in detecting model failures, and efficiency, i.e., the time required for detecting the failures. Our results show that the SBST framework successfully identified a failure-revealing test case for 66.7% of our model versions (14 out of 21). The SBST framework required 245.9s and 3.8 iterations (on average) to detect the failure-revealing test cases. Only the last version of the model can pass the most complex and generic scenario we tested.

In summary, this work addresses the following problems:

P1 it assesses the usefulness of SBST in detecting failure-revealing test cases for the development of a complex CC for an *industrial simulator*;

P2 it assesses the usefulness of SBST in detecting failure-revealing test cases for *drivability requirements*, an important and large category of automotive requirements;

P3 it assesses the usefulness of SBST driven by *Simulink® Test Blocks*, that are standard tools for test case specification in Simulink®.

These research problems are relevant to the industry and motivated by forthcoming industrial challenges. Understanding the usefulness of SBST in detecting failure-revealing test cases for CC development (**P1**) enables automotive industries to assess how beneficial SBST techniques are and helps them to evaluate how to use them within their development processes. Understanding the usefulness of SBST in detecting failure-revealing test cases for drivability requirements for the CC (**P2**) enables automotive industries to know how frequently these requirements are violated compared to the functional requirements. This information helps them understand how to prioritize and analyze different requirements across the CC development. Finally, understanding the usefulness of SBST driven by Simulink® Test Blocks (**P3**) helps automotive industries assess how beneficial this technique is in developing complex automotive systems.

This work is organized as follows. Section 2 describes our automotive case study. Section 3 summarizes the CC development activities and the application of the testing framework. Section 4 presents SBST and HECATE, the SBST framework considered in this work. Section 5 presents our evaluation methodology and results. Section 6 discusses results, lessons learned, and the improvement on the state of practice. Section 7 presents related work. Section 8 concludes the work.
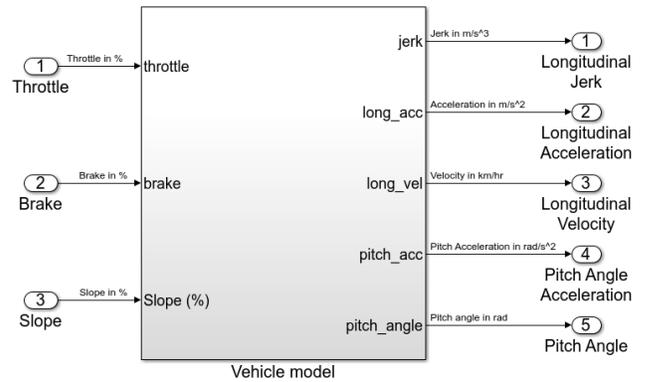

Figure 1: MARCdrive simulator.

Figure 2: Simulink® model under analysis.

2 CASE STUDY

This section presents our automotive case study: it describes the controlled system (Section 2.1) and its functional and drivability requirements (Section 2.2).

2.1 Controlled System

Our case study uses the VI-CarRealTime (VICRT) Simulator [77]. The controlled system is a vehicle traveling on the road: the inputs and outputs of the vehicle relevant to our study are schematized in Figure 2. The inputs are the throttle, brake, and slope percentages that respectively assume values within the ranges $[0, 100]\%$, $[0, 100]\%$, and $[-5, 5]\%$. The outputs are the longitudinal jerk, acceleration, and velocity, and the pitch angle and acceleration. The goal of the software engineer is to design a cruise controller (CC) that measures the longitudinal velocity, the pitch angle, and slope percentages and acts on the throttle and brake to ensure the satisfaction of the functional and drivability requirements (see Section 2.2).

During the CC development, the software engineer can use SBST to automatically search for failure-revealing test cases, i.e., test cases that show a violation of the functional and drivability requirements. As we will describe later (Section 5), we performed software-in-the-loop (SIL) and hardware-in-the-loop (HIL) simulations to test the CC model. For SIL simulations, the vehicle is represented by a Simulink® model provided by the VICRT framework. This model contains 1125
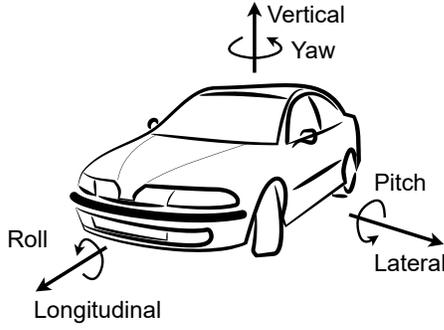
Figure 3: Axes used for the definition of the drivability requirements.

Simulink® blocks and relies on an XML file produced by VICRT. The XML file contains all the vehicle properties (e.g. motor speed-torque curves, the tire properties), used by the Simulink® model to simulate faithfully the vehicle dynamics.

For HIL simulations, we considered the MARCdrive simulator [24] from the McMaster Automotive Resource Centre (MARC) [55] shown in Figure 1. This simulator integrates VICRT with the model of a Sedan Car, i.e., a passenger car with separate compartments for the engine, passengers, and cargo. It is a high-fidelity, customizable simulator that enables engineers to test their projects. The projections of the road and surrounding environment on the 210-degree screen, the sound system, and the active steering wheel, brakes, driver's seat, and seatbelt enable the driver to perceive the vehicle dynamics (speed and acceleration). The VICRT calculates the dynamics of the car and provides feedback to the driver.

For our experiments, we considered the vehicle traveling on a straight trajectory. We analyzed this trajectory since it is commonly considered for acceleration tests by automotive CC developers and vendors [52, 54].

2.2 Functional and Drivability Requirements

We analyzed the functional (F1) and drivability requirements (D1, D2, D3) from Table 1. These requirements are largely considered in the research literature; the interested reader can consult the following publications: [65] for F1, [41, 60, 64] for D1, [69] for D2, and [52] for D3.

The functional requirement F1 requires the vehicle speed to reach and stay within 3 km/hr of the desired velocity after 30 seconds of the velocity being set. We selected the threshold of 30 seconds as it allows for an operating range of 100 km/hr for the CC while still meeting all drivability requirements.

The drivability requirements limit the longitudinal acceleration and jerk, and the pitch acceleration (see Figure 3) within given thresholds ranges.

The drivability requirement D1 requires the longitudinal acceleration to remain within the range $[-3.5\frac{m}{s^2}, 5\frac{m}{s^2}]$ (i.e., $[-0.36g, 0.51g]$). The threshold values for the longitudinal acceleration were defined after testing different acceleration and braking scenarios on two authors using the MARCdrive simulator. The selected range includes the expected upper limit of $1.47\frac{m}{s^2}$ for acceptable longitudinal acceleration in public transportation [41].

Table 1: Requirements for our controller the VI-CarRealTime Simulator controller.

ID	Description
F1	After every change, the system shall reach and stay within $3\frac{km}{h}$ of the desired speed after 30 seconds.
D1	The longitudinal acceleration of the vehicle shall not exceed $-3.5\frac{m}{s^2}$ and $5\frac{m}{s^2}$.
D2	The absolute value of the jerk of the vehicle shall not exceed $10\frac{m}{s^3}$.
D3	The absolute value of the acceleration of the vehicle on the pitch axis shall not exceed $3\frac{rad}{s^2}$.

The drivability requirement D2 requires the absolute value of the acceleration of the vehicle on the pitch axis to not exceed $3\frac{rad}{s^2}$. This value was determined experimentally on the MARCdrive simulator, in the same way as the longitudinal acceleration thresholds. We verified that these values are reasonable by consulting relevant studies in the field (e.g., [27]).

The drivability requirement D3 requires the longitudinal jerk (i.e., the acceleration change ratio over time) to be lower than $10\frac{m}{s^3}$. This value is considered comfortable for the driver and passengers [42].

After defining the requirements of the system, the development of the CC began.

3 DEVELOPMENT OF CRUISE CONTROL

The CC model was developed by two of the authors: a mechatronics fourth-year bachelor student, and a first-year Ph.D. student in software engineering with large experience in the development of controllers for CPS. The development activity required approximately 300 hours. This includes the time required to develop the CC versions and the time needed for testing them. In this section, we summarize the characteristics of the different versions of the CC and present its final version. The testing activity will be discussed in Section 5.

Table 2 lists the different versions of the model of the CC. For each version of the model, the table provides a description of the changes introduced in that version, the number of inputs, and the number of blocks in the model. Finally, the table shows which requirements were considered during the testing activities (discussed in Section 4 and Section 5) on each version: the functional requirement F1 for the first two versions of the model, and the requirements F1, D1, D2, D3 for the remaining versions.

The model is developed incrementally, and new features are added to the CC until all the requirements are satisfied. For example, version 1.0 is a PID (Proportional, Integral, Derivative) software controller that changes the vehicle's throttle depending on the difference between the velocity and the desired velocity. The higher the difference, the larger the throttle. Version 2.0 also acts on the vehicle brakes to reduce the vehicle speed when the desired velocity is lower than the actual velocity. Version 3.0 adds a smoothing algorithm to smooth non-continuous outputs from the PID controller.

The final version of the CC contains 322 blocks. Figure 4 presents the high-level structure of the final version of the CC. The CC consists of five main subsystems detailed as follows.
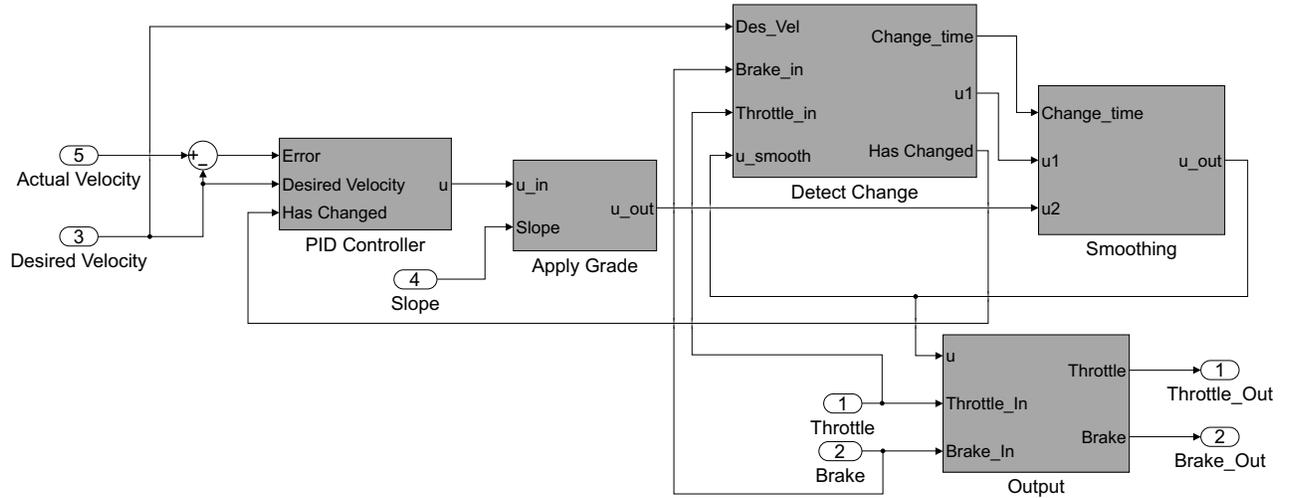
Figure 4: Cruise Controller structure in the latest version.

(1) *PID Controller.* Calculates the difference between the desired velocity and actual velocity of the system, and outputs the sum of the error, its derivative, and its integral, each multiplied by a certain gain. A proportional gain is also added directly to the desired velocity to compensate for different speeds requiring different amounts of throttle to maintain. This subsystem also uses changes detected in the *Detect Change* subsystem to reset the integral portion of the controller.

(2) *Apply Grade.* Modifies the output of the PID controller depending on the slope of the road: it respectively increases or decreases depending on whether the slope is uphill or downhill. This operation compensates for more or less brake and throttle being necessary for those conditions.

(3) *Detect Change.* Monitors changes in the driving mode that can cause discontinuous behaviors in the CC and takes appropriate actions. Monitored changes include (a) the driver releasing the throttle and brake pedals causing the CC to automatically take over, and (b) the desired velocity changing.

(4) *Smoothing.* Creates a smooth transition between the old and the new throttle and brake values after a change in the driving mode.

(5) *Output.* Splits the throttle and brake signals and sends them to the vehicle. Deactivates the cruise control if either throttle or brake applied by the driver is above 5% of their maximum value.

SBST was used to check for requirement violations on the different versions of the CC, including its final version.

4 SEARCH-BASED SOFTWARE TESTING

This section presents an overview of SBST tools for Simulink® models (Section 4.1), and HECATE [35], the framework we selected in this work (Section 4.2).
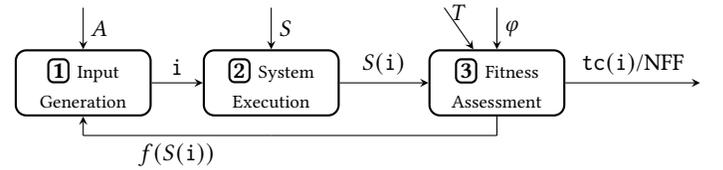

Figure 5: Overview of an SBST framework.

4.1 SBST for Simulink® Models

The main steps performed by an existing SBST for Simulink® models in searching for failure-revealing test cases are depicted in Figure 5: squared boxes represent their steps, arrows connect subsequent steps describing their inputs and outputs, arrows with no source and destination are the inputs and outputs of the SBST framework.

Simulink® SBST frameworks typically rely on four inputs: a model of the system to be tested (S); an assumption on the system inputs (A), a time budget (T), and a requirement (φ). Their output is a failure-revealing test case (tc(i)) or an indication that no failure-revealing test case was found (NFF — No Failure Found) within the time budget.

The SBST frameworks iteratively repeat these steps to search for a failure-revealing test case:

1. *Input Generation*: generates an input (i) for the system model (S) that satisfies the assumption (A);
2. *System Execution*: simulates the system model (S) for an input (i) and obtains a system execution ($S(i)$).
3. *Fitness Assessment*: computes the fitness value ($f(S(i))$) for the system execution ($S(i)$) and assesses whether the fitness value is below a threshold value.

A test case (tc(i)) is failure-revealing if: (a) the corresponding input (i) satisfies the assumption (A), and (b) the fitness value ($f(S(i))$) is smaller than a threshold value. Typically, the threshold value is

Table 2: ID, Description, Requirements ID (Req), Number of Inputs (#In) and Number of Blocks (#Bck) of the different versions of our controller.

ID	Description	#In	#Bck*	#Req
1.0	Initial version of the PID model for the controller.	1	71	F1
2.0	Applies brakes when necessary and deactivates the CC by applying the throttle or brake.	1	94	F1
2.1	Introduces drivability requirements.	1	94	F1,D1,D2,D3
3.0	Adds smoothing system by limiting the derivative of outputs from the PID controller.	1	142	F1,D1,D2,D3
3.1	Introduces Test Sequence 2.	1	142	F1,D1,D2,D3
3.2	Introduces Test Sequence 3.	1	142	F1,D1,D2,D3
4.0	Adapts the velocity to the slope of the road and changes the smoothing system.	2	261	F1,D1,D2,D3
5.0	Eliminates the spikes in jerk and improves response in downhill conditions.	2	348	F1,D1,D2,D3
6.0	Adds slope detection system within the vehicle, allowing for it to be used within the control system.	2	379	F1,D1,D2,D3
6.1	Modifies the proportional gain to consider the slope of the road.	2	380	F1,D1,D2,D3
6.2	Makes smoothing more aggressive and changes the conditions that would trigger it.	2	402	F1,D1,D2,D3
6.3	The signal from the PID controller is smoothed before being applied to throttle or brake.	2	342	F1,D1,D2,D3
6.4	Changes the sigmoid smoothing equation to have an initial derivative closer to 0.	2	335	F1,D1,D2,D3
6.5	Uses cubic splines that hit and stay at zero for the throttle and brake profiles.	2	332	F1,D1,D2,D3
6.6	Removes all possibility of brake and throttle signals being on at the same time.	2	330	F1,D1,D2,D3
7.0	Prevents the throttle signal from ever going below 5, which causes a large spike in jerk.	2	339	F1,D1,D2,D3
7.1	Limits the maximum power of brakes preventing negative acceleration violations.	2	339	F1,D1,D2,D3
7.2	Changes the brake profile to a degree 5 polynomial, making it overall smoother.	2	329	F1,D1,D2,D3
7.3	Resets in the integral in the PID controller to prevent large overshoots after long run times.	2	337	F1,D1,D2,D3
7.4	Introduces Test Sequence 5.	2	337	F1,D1,D2,D3
7.5	Introduces Test Sequence 6.	2	337	F1,D1,D2,D3

* This figure counts the VICRT mask as a single block, though it actually contains 1125 blocks.

(a) Parameterized Test Sequence Block.

(b) Test Assessment Block.

Figure 6: An example of a test sequence and a test assessment block for our cruise controller.

0 since the fitness value is negative if the requirement is violated and positive otherwise (e.g., [29, 31, 59]).

Several tools for Simulink® models implement the framework from Figure 5, such as ARIsTEO [58], ATheNA [34], Breach [28], FalCAuN [78], falsify [80], FalStar [30], ForeSee [81], STGEM [63], S-TaLiRo [10], Ψ-TaLiRo [72], and HECATE [35]. In this work, we used HECATE [35] to support the development of our cruise control model since, unlike the other tools, it supports Simulink® Test Blocks that are commonly used for testing CPS models.

4.2 HECATE

HECATE supports Simulink® Test Blocks (i.e., Test Sequences and Test Assessment blocks). Specifically, to support SBST, HECATE

supports Parameterized Test Sequences [35]: Test Sequences extended with parameters that are used for automated test case generation. Parameterized Test Sequences are detailed later in this section. Figure 6 presents an example of a Parameterized Test Sequence (Figure 6a) and a Test Assessment (Figure 6b) block for requirement F1 of our cruise controller. The Test Sequence block describes the input of the system; the Test Assessment block describes the requirement to be satisfied. Both the Test Sequence and the Test Assessment are Test Blocks; therefore, they share part of their syntax. The syntax and semantics of Test Sequence and Test Assessment blocks are summarized in the following paragraphs. An extensive description can be found online [70, 71].

Test Blocks consist of *test steps* connected by *transitions*. The Parameterized Test Sequence block in Figure 6a and the Test Assessment block in Figure 6b contains three (**Start**, **Step_0**, **Step_1**) and six (**CCActive**, **RecentChange**, **ThirtySecondThreshold**, **VelocityChange**, **DriverActive**, **Buffer**) test steps respectively. Test steps are hierarchically organized. For example, the test step **RecentChange** of Figure 6b is nested within the test step **CCActive**. At each time instant, the Test Blocks are in exactly one 'child' test step and its 'ancestors' in the hierarchy. For example, if the Test Assessment in Figure 6b is in the child test step **VelocityChange**, it is also in all of its ancestors, namely **CCActive**. Whenever a Test Block enters a 'non-child' test step, it also enters its first child test step. For example, if the **CCActive** test step of Figure 6b is entered, its child test step **RecentChange** is also entered.

Transitions define how a Test Block switches between test steps. They connect a source and a destination test step. They are labeled with a Boolean formula representing a condition for the transition to be fired. For example, transitions of the Test Sequence block from Figure 6a specify how the Test Sequence switches between the test steps **Step_0** and **Step_1**. Whenever a non-child test step is exited, so are all of its 'descendants'. For example, if the Test Assessment of Figure 6b is in the test step **CCActive** and the value of the brake and the throttle become greater than 5, the **CCActive** test step is **RecentChange** is also left.

Test steps from Test Sequence and Test Assessment blocks differ in their purpose. Test Sequences define the values assumed by the input signals. For example, the test step **Start** of the Test Sequence in Figure 6a assigns the value 0 to the variable throttle. Unlike Test Sequence blocks, Test Assessment blocks contain verification statements, e.g., **verify**, which check whether a logical expression evaluates to TRUE or FALSE. For example, the statement **verify**(abs(desired_velocity-velocity)<=3)) of the test step **ThirtySecondThreshold** in Figure 6b checks whether the absolute difference between the desired velocity and the velocity of the vehicle is lower than 3 km/h.

A *test case* is made by a Test Sequence block and a Test Assessment block. The Test Sequence block generates a test input, i.e., a set of input signals, that are fed into the Simulink® simulator. A test input is then executed by simulating the model for the input signals associated with the test sequence. The Test Assessment block evaluates if the output signals of the model lead to a violation of the expressions of its **verify** statements. A Test Assessment block is typically associated with the requirements of the system. For example, the Test Assessment block from Figure 6b checks whether the conditions from requirement D1 holds. A test case is failure-revealing if the test input generated from the Test Sequence block leads to a set of output signals that violate the conditions specified by the Test Assessment block.

Parameterized Test Sequences enable the addition of search parameters to the Test Sequence block. These parameters are represented by variable names preceded by the keyword Hecate. For example, the Test Sequence block from Figure 6a contains three search parameters: Hecate_desVel1, Hecate_desVel2, and Hecate_Transition1. These search parameters (detailed in Table 3) represent the desired initial velocity, and final velocity, and the time for which the initial velocity is maintained.

Table 3: Test Sequence Parameters definition and range.

Parameter	Description	Min	Max
*des_vel1**	Desired initial velocity.	50	150
*des_vel2**	Desired intermediary velocity.	50	150
des_vel3	Desired final velocity.	50	150
transition1	Time for the initial velocity.	30	40
transition2	Time for the intermediary velocity.	0	20
slope	The slope of the road.	-4	4
verShift	Average value of the slope.	-1	1
period	Period of the sinusoidal slope.	30	200
horShift	Horizontal shift of the slope.	0	π

des_vel1, *des_vel2*, *des_vel3* are in km/h. *transition1*, *transition2*, *verShift*, *period*, *horShift* are in s. *slope* and *verShift* are %.
* For the first Test Sequence (TS1 in Table 4), *des_vel1* and *des_vel2* are in the range [0, 100]km/h, since the CC is activated with zero initial velocity.

HECATE is an automated SBST framework that implements the framework from Figure 5. For the input generation phase (①), HECATE iteratively generates new Test Sequences by assigning values to the parameters of the Parameterized Test Sequences. For the system execution phase (②), HECATE relies on the Simulink® simulator to produce the output signals associated with a Test Sequence block. Finally, for the fitness assessment phase (③), HECATE relies on a specific translation that converts Test Assessment blocks into fitness functions (the interested reader can refer to [35] for additional information). Phases (①, ②, and ③) are executed until a failure revealing test case is detected, or a time budget (number of test cases evaluated) is exceeded. In the first case, HECATE returns the search parameter values associated with the failure-revealing test case. Otherwise, it returns the NFF value.

We use HECATE to assess the usefulness of SBST.

5 EVALUATION

Our evaluation estimates the usefulness of SBST in developing CPS by assessing its support for developing a complex CC model for an industrial simulator and answering two research questions:

RQ1: How *effective* is SBST in producing failure-revealing test cases? (Section 5.1)

RQ2: How *efficient* is SBST in producing failure-revealing test cases? (Section 5.2)

RQ1 assesses how helpful SBST is in detecting model failures. RQ2 assesses the time required for detecting these failures.

5.1 Effectiveness (RQ1)

To assess the effectiveness of SBST in producing failure-revealing test cases, we proceeded as follows.

Methodology. We ran our SBST framework for each version of the CC model from Table 2 using the SIL simulations. We set the cut-off value of 20 iterations: since a single model simulation requires 100s, this value ensures that the test case generation takes approximately 30min (at maximum 34min). If our test case generation

Table 4: Test Sequences considered for testing the model: Identifier (ID), Number of Steps (#S), Usage of the Slope Parameter (SP), Manual Startup (MS), Velocity Range (VR), Simulation Time (T), and Description.

ID	#S	SP	MS	VR	T	Description
TS1	1	U	No	0-100	70s	CC starts from stationary and has one step.
TS2	1	U	Yes	50-150	70s	Driver accelerates to a speed above 50 km/h before activating CC with one step.
TS3	2	U	Yes	50-150	100s	Driver accelerates to a speed above 50 km/h before activating CC with two steps.
TS4	1	C	Yes	50-150	70s	Driver accelerates to a speed above 50 km/h before activating CC with one step on a constant slope.
TS5	2	C	Yes	50-150	100s	Driver accelerates to a speed above 50 km/h before activating CC with two steps on a constant slope.
TS6	2	S	Yes	50-150	100s	Driver accelerates to a speed above 50 km/h before activating CC with two steps on a sinusoidal slope.

U: Unused, C: Constant, S: Sinusoidal

Table 5: Parameters used in each Test Sequence.

Parameter	TS1	TS2	TS3	TS4	TS5	TS6
des_vel1	✓	✓	✓	✓	✓	✓
des_vel2	✓	✓	✓	✓	✓	✓
des_vel3			✓		✓	✓
transition1	✓	✓	✓	✓	✓	✓
transition2			✓		✓	✓
slope				✓	✓	✓
verShift						✓
period						✓
horShift						✓

framework returned a failure-revealing test case, we assessed the test case to find and fix the source of the error. We iteratively modified the CC model, and re-ran the test case generation framework, until the test case generation framework was not able to produce any failure-revealing test case for that requirement. When the SBST framework was not able to produce any failure-revealing test case, we proceeded by extending the test scenario (e.g., introducing slope as a parameter). In our experiments, the input generation algorithm was Uniform Random [13], since 20 iterations do not enable other optimization algorithms to converge to a good solution.

We ran our SBST framework by considering six different Test Sequences detailed in Table 4: as new features were added to the model, we considered more complex Test Sequence blocks to trigger these features. For example, the Test Sequence *TS1* (detailed in Figure 6a) activates the CC immediately, with an initial vehicle speed of 0 km/h (no Manual Startup). We considered a more realistic Test Sequence (*TS2*), which had the driver accelerate to 50 km/hr before activating the CC (with Manual Startup), when the CC could switch itself off when the driver applied brake or throttle. Table 6 reports the Test Sequences considered for the different versions of the model. The Test Sequences rely on the Test Sequence parameters from Table 3. A Test Assessment block models the requirements of the CC. The SBST algorithm searches values for the parameters of the Test Sequences that violate the conditions specified by the Test Assessment blocks, i.e., lead to requirement violations. If a violation is detected, the corresponding Test Blocks (Test Sequence and Test Assessment) of the failure-revealing test case are returned. We verified the failure-revealing test cases also on the hardware platform (see Figure 1) by running them with the HIL simulator.

We analyzed how many versions of the CC model the SBST framework was helpful by returning a failure-revealing test case. We also verified that the HIL and SIL simulations yielded consistent results for the failure-revealing test cases.

Results. Table 6 presents our results. The table reports the version of the model (ID) and the Test Sequences (TS#) considered for that version of the model. For each version, the table reports the number of iterations (#IT) required to identify the failure-revealing test case, which requirement was not satisfied, and a description of the identified failure. The SBST framework returned a failure-revealing test case for 66.7% of the versions of our model (14 out of 21). The versions of the model that did not yield failing test cases were then tested with 50 iterations of the Simulated Annealing algorithm [2]. Of the models that returned a failing test case, 42.9% (6 out of 14) were due to functional violations, 21.4% (3 out of 14) were due to drivability violations, and 35.7% (5 out of 14) were due to both.

For example, for version 1.0 of the model, the SBST returned a failure-revealing test case for the functional requirement F1. The test case revealed that the controller did not act on the vehicle brakes to reduce the speed: when the desired speed was lower than the speed of the car, the vehicle's aerodynamic resistance was insufficient to ensure the vehicle reached the desired speed within 30s. The failure was solved by version 2.0 of the model. For version 2.1 of the model, the SBST returned a failure-revealing test case for the drivability requirement D1. Figure 7 shows a snapshot of our simulation, revealing that the CC acts on the brake and the throttle by leading to a longitudinal acceleration of $-11.1 \frac{m}{s^2}$ that exceeds the threshold values ($-3.5 \frac{m}{s^2}$ and $5 \frac{m}{s^2}$) specified by the requirement D1. The corresponding video is available online [3].

> **RQ1 - Effectiveness**
>
> The answer to RQ1 is that, for our automotive model, SBST returned a failure-revealing test case for 66.7% of our model versions (14 out of 21). The failure-revealing test cases enabled the identification of the failures described in Table 6. The test cases revealed a violation of the drivability requirements for 21.4% of the cases.

5.2 Efficiency (RQ2)

To assess the efficiency of our SBST framework in producing failure-revealing test cases, we proceeded as follows.

Table 6: Model failures identified by analyzing the failure-revealing test cases.

ID	TS#	#IT	F1	D1	D2	D3	Description of Failure
1.0	1	1	✗	N.T.	N.T.	N.T.	The controller does not act on the brakes to decrease the speed of the vehicle.
2.0	1	> 20	✓	N.T.	N.T.	N.T.	No failure; introduced drivability requirements.
2.1	1	9	✓	✗	✗	✓	The controller discontinuously applies brakes or throttles.
3.0	1	> 20	✓	✓	✓	✓	No failure detected; introduced test sequence 2.
3.1	2	> 20	✓	✓	✓	✓	No failure detected; introduced test sequence 3.
3.2	3	> 20	✓	✓	✓	✓	No failure detected; introduced slope parameter.
4.0	4	1	✗	✓	✗	✓	Numerical errors in the VICRT mask cause random spikes in jerk during run time.
5.0	4	1	✗	✓	✓	✓	Large negative/positive slopes prevent the vehicle from timely reaching the desired velocity.
6.0	4	1	✗	✗	✓	✓	Large changes in desired velocity cause the vehicle to violate acceleration requirements.
6.1	4	8	✗	✓	✗	✓	Fails to solve the previous problems.
6.2	4	2	✓	✓	✗	✓	Fails to solve the previous problems.
6.3	4	1	✗	✓	✗	✓	Fails to solve the previous problems.
6.4	4	1	✗	✗	✗	✓	Fails to solve the previous problems.
6.5	4	2	✓	✗	✓	✓	Fails to solve the previous problems.
6.6	4	2	✗	✓	✓	✓	Fails to solve the previous problems.
7.0	4	7	✗	✓	✓	✓	Large negative changes in desired velocity are impossible to reach without violating D1.
7.1	4	14	✗	✓	✓	✓	Very specific test cases marginally fail functional requirements.
7.2	4	4	✗	✓	✓	✓	Fails to solve the previous problem.
7.3	4	> 20	✓	✓	✓	✓	No failure detected; introduced test sequence 5.
7.4	5	> 20	✓	✓	✓	✓	No failure detected; introduced test sequence 6.
7.5	6	> 20	✓	✓	✓	✓	No failure detected.

✗: Requirement not met; ✓: Requirement met; N.T.: Requirement not tested.

Figure 7: HIL simulation of a test case where the car breaks the drivability requirements.
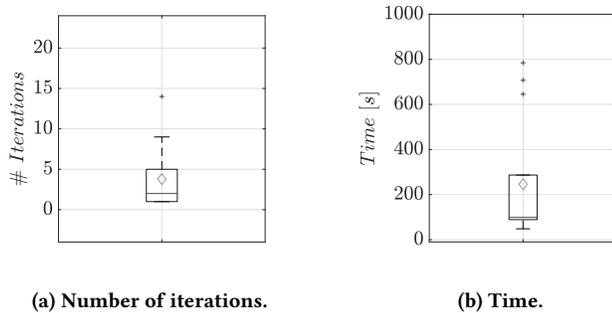

(a) Number of iterations. (b) Time.

Figure 8: Iterations and time of our SBST framework.

Methodology. We considered the runs from RQ1 returning a failure-revealing test case. We analyzed the number of iterations and time required to detect the failure-revealing test case.

Results. The box plots in Figure 8 report the number of iterations (Figure 8a) and the time (Figure 8b) required to produce the failure revealing test cases. Our SBST framework required, on average 3.8 (*min*=1, *max*=14, *StdDev*=3.9) iterations and 245.9s (*min*=48.2s, *max*=784.5s, *StdDev*=262.2s) to generate these test cases.

RQ2 - Efficiency

The answer to RQ2 is that, for our automotive model, the SBST framework required on average 3.8 iterations (*min*=1, *max*=14, *StdDev*=3.9) and 245.9s (*min*=48.2s, *max*=784.5s, *StdDev*=262.2s) to return the failure-revealing test cases.

5.3 Discussion and Threats to Validity

Our results confirm the usefulness of SBST in detecting failure-revealing test cases of a complex CC for an industrial simulator (**P1**): the SBST returned a failure-revealing test case for the majority (66.7%) of our model versions in practical time (245.9s ≈ 4min). They also confirm the usefulness of SBST in detecting failure-revealing test cases for drivability requirements (**P2**): 21.4% of the failure-revealing test cases showed a violation of a drivability requirement. Finally, our results confirm the usefulness of SBST driven by Simulink® Test Blocks (**P3**): we could express all the requirements of the CC using the Test Assessment blocks, and our search space using Parameterized Test Sequences.

Our results are subject to the following threats to validity.

The automotive benchmark model, requirements, and controller versions we analyzed could threaten our results' *external validity*

since different benchmark models, requirements, and controllers can lead to different results for the effectiveness and efficiency of the SBST. The fact that our industrial model is used by many automotive companies (e.g., Brembo [1]) and taken from the VI-CarRealTime simulator, the functional and drivability requirements come from the research literature (e.g., [65], [41, 60, 64], [69], [52]), and the CC model is a large and complex model developed in eight months mitigate this threat. The number (1461) of blocks of the final model (system under test plus CC) is a reasonable approximation to a small to mid-sized realistic industrial model [21].

The values of the configuration parameters of our SBST could threaten the *internal validity* of our results: a higher number of iterations for our SBST framework or a different search algorithm can produce different results [36]. To mitigate this threat, for the versions of the CC for which the SBST did not return any failure-revealing test case, we increased the number of iterations to 50 and considered an additional search algorithm (Simulated Annealing [2]). This configuration of our SBST framework also did not return any failure-revealing test case for these versions of the CC.

6 DISCUSSION

This section reflects on the lessons learned (Section 6.1), the improvement on the state of the practice (Section 6.3), and the generality of our results (Section 6.2).

6.1 Lessons Learned

The results of our experimental evaluation confirm the usefulness of SBST in detecting failure-revealing test cases of a complex CC (**P1**), for drivability requirements (**P2**), and Simulink® Test Blocks (**P3**).

In addition, we learned many lessons from our experimental evaluation: we report on five of these lessons (L1-L5).

Running many simulations is compute-intensive (L1). Testing our automotive case study requires significant computational time. For our automotive system, executing a test case takes around 100s. Therefore, running 20 iterations of our SBST framework takes approximately 34min. This time may be prohibitive for many applications that require executing a vast number of test cases that need to be executed in-house for confidentiality reasons or due to specific configurations of the machines that are difficult to replicate on large computing platforms.

The input domain of our model is large (L2). In our testing activity, we considered the vehicle traveling in a straight trajectory. We also assumed that no other vehicles were present on the road, and the CC can only be set to speeds between 50 km/h and 150 km/h. Despite these assumptions, the input domain is infinite: the values of the slope percentage (represented as a real variable) can be changed (across the simulation time) in an infinite number of ways. In addition, even though our assumptions are reasonable for our experiments, large testing campaigns may require considering other input configurations (e.g., different input shapes), leading to additional problems for the applicability of SBST. The large input domain poses the engineers with trade-off decisions: on the one hand, the engineer wants to increase the number of simulations to increase the number of inputs that are analyzed; on the other hand, since the automotive model is compute-intensive (L1), a limited number of simulations can be executed.

HIL testing is laborious (L3). HIL requires time and effort. Configuring our hardware platform for our experiments required significant effort. Starting and shutting down the simulator requires 10 minutes and the channels to transmit the signals to and from the hardware must be checked manually for each model version (24 input signals and 21 output ones). Running experiments using the hardware platform may also lead to unpredicted problems: for example, while running our experiments, the light of one of the projectors of our simulator stopped working, and we had to replace it. Replacing the light of the projector required a few weeks. In addition, HIL testing activities could only be performed within the McMaster facilities to access the hardware platform, and we had to ensure the presence of humans monitoring the simulator's behavior as mandated by our laboratory regulations. The presence of a human limits the number of HIL tests that can be executed on the hardware platform that, therefore, need to be carefully selected.

Results of HIL and SIL simulations may differ (L4). We notice discrepancies in the results of HIL and SIL simulations for some of the test cases. Specifically, HIL simulations introduce some noise (with non-zero average) on the throttle pedal, brake pedal, and steering wheel angle signals that were not present in the SIL simulations. Furthermore, the HIL simulations have a sampling time of 10ms, while SIL uses 1ms. This causes a difference in the way the jerk is computed and produces much lower values for the HIL results. These minor discrepancies are enough to change the test verdict in some of the test cases. We will carefully investigate the discrepancies between HIL and SIL simulations and how to mitigate them in future work.

SIL testing activities need to be automated (L5). Manually executing the SBST framework after any change in the model is laborious and needs to be automated. Integrating the SBST activity in a continuous pipeline that automatically fetches the model and runs the SBST activity after each version of the model is produced can alleviate this challenge.

The challenges we identified in our automotive case study are general and apply to other CPS domains, as argued in the following.

6.2 Generality of the Results

To assess the generality of our results and lessons learned, we critically compared them to those from the literature.

For the usefulness in detecting failure-revealing test cases for a complex CC (**P1**), we do not expect that the number of model versions in which the SBST returned a failure-revealing test case will be the same for other CC, automotive models, and CPS. However, we believe that these findings apply to other types of CPS. Empirical studies from the literature confirm this hypothesis in other domains (e.g., space [58], automotive [32, 68], biomedical [16], medical [12, 16, 22]).

For the usefulness in detecting failure-revealing test cases for drivability requirements (**P2**), we do not expect that the percentage of test cases that reveals a violation of the drivability requirements will match the one obtained by analyzing other CC models since the results of SBST strongly depend on the model that is analyzed. However, we believe that SBST will help detect failure-revealing test cases for the drivability requirements of other CC models. Future empirical studies will confirm or refute our hypothesis.

For the usefulness in detecting failure-revealing test cases for Simulink® Test Blocks (**P3**), we do not expect the values for the effectiveness and efficiency to match the one obtained by analyzing other systems, since the results depend on the model and the Test Blocks considered by the SBST framework. However, we believe that the SBST framework is also useful to analyze Simulink® Test Blocks that refer to other CPS. The results provided by HECATE [35] confirm our hypothesis. The authors of this work have shown that SBST was useful in detecting failure-revealing test cases for a large set of benchmarks consisting of 16 Simulink® models from different domains (automotive, energy, biomedical, avionics, domotics), including three developed by Lockheed Martin (EU, NNP, TUI) [56].

We believe that the lessons we learned (L1, L2, L3, L4, and L5) generalize to other types of CPS. There are several additional studies from the literature confirming these hypotheses (i.e.,[58, 76] for L1, [46, 47] for L2, and [57, 67] for L3 and L4).

We conclude that our results are likely generalizable and may also apply to other CPS domains. Additional empirical studies and experimental analysis may confirm or confute our hypothesis across the different domains. In the following section, we will also argue that our results significantly improve the state of the practice.

6.3 Improvement on the State of the Practice

Our results significantly improve the state of practice.

Our results show that SBST effectively supports the development of a complex CC for an industrial simulator (**P1**). Our results improve the state of practice by providing empirical data, evidence, and reflections related to using SBST to develop CC. Our findings, i.e., the percentage of model versions in which the SBST could return a failure-revealing test case and the number of iterations and time needed to compute them, provide automotive companies with empirical data that can help them decide when and how to use SBST and estimate the resources (e.g., time and computational resources) needed to conduct this activity.

Our results show that SBST effectively supports the analysis of drivability requirements (**P2**). We argued (Section 1) that drivability requirements are critical for automotive applications. Despite their importance, the research literature does not extensively assess the usefulness of SBST frameworks for these requirements. This work evaluates the usefulness of SBST for drivability requirements experimentally via a case study supported by an industrial simulator. Our results improve the state of practice by showing that SBST effectively finds failure-revealing test cases for drivability requirements. They also show that the test cases revealed a violation of the drivability requirements for 21.4% of the failure-revealing test cases. This result suggests that automotive industry needs to extensively consider these requirements since, for a significant number of cases, the failure-revealing test cases refer to these requirements rather than the functional requirements of the CC. This finding will impact the automotive domain, supporting a more extensive adoption of SBST frameworks driven by drivability requirements.

Our results show that SBST driven by Test Blocks is useful (**P3**). Although existing work has shown the effectiveness of SBST for Simulink® models, the only framework that supports Test Blocks is HECATE. It was evaluated by considering a benchmark consisting of 16 Simulink® models from different domains. However, this work does not assess SBST for the end-to-end development of a CPS, i.e., the benchmark does not include a model for which all the versions produced during the software development are available. Unlike the work introducing HECATE, we assessed the effectiveness of SBST driven by Test Blocks during the end-to-end development of a CC for an automotive system supported by an industrial simulator. Our results improve the state of practice by showing that SBST driven by Test Blocks effectively supports developer activities and paves the way for adopting this framework within the industry and for more empirical studies assessing the advantages and limitations of the usage of SBST driven by Test Blocks in the end-to-end development of CPS.

7 RELATED WORK

Several works assessed the usefulness of SBST in detecting failure-revealing test cases for the development of CPS (e.g., [7, 12, 22, 49, 50, 76]) and CCs (e.g., [38, 79]). Unlike these works, we assessed the usefulness of SBST by considering the end-to-end development of the CC and used an industrial simulator for our simulation.

Several other studies have used the VI-CarRealTime industrial simulator. For example, this simulator was used to analyze alternative steering mechanisms [73], the relationship of steering and stability [40], and other aspects of automated driving [15, 23, 25, 33, 62]. However, none of these studies used the simulator to assess the usefulness of SBST.

Although drivability requirements are extensively considered during the design of CC (e.g., [9, 11, 18, 52, 61, 66]), the assessment of SBST techniques typically focuses on the functional requirements of the CC (e.g., [37, 46, 75]). Our results showed the importance of considering the drivability requirements, i.e., for a significant number of the generated failure-revealing test cases (21.4%), the failure caused a violation of the drivability requirements.

Finally, based on our knowledge, the only work that assesses the usefulness of SBST driven by Simulink® Test Blocks is the one proposed by HECATE [35].

8 CONCLUSION

We assessed the usefulness of SBST in generating failure-revealing test cases for the functional and drivability requirements of a CC of a vehicle. We used an SBST framework during the end-to-end development of a complex CC developed using Simulink®. We performed SBST relying on a framework that supports Simulink® Test Blocks. We performed both SIL and HIL testing using an industrial simulation environment widely used in the automotive domain.

The empirical results of our experimental evaluation confirm that SBST is useful in detecting failure-revealing test cases of a complex CC, for drivability requirements, and Simulink® Test Blocks. We presented the lessons we learned from our experimentation, reflected on the generality of our results, and discussed how our results improve the state of practice.

DATA AVAILABILITY

A replication package containing all of our data, test results, and videos recorded during HIL simulations is publicly available [3].

REFERENCES

[1] 2022. *VI-GRADE Announces Installation of Latest Generation Compact Driving Simulator at Italian Braking Systems Manufacturer, Brembo.* https://www.vi-grade.com/en/about/news/vi-grade-announces-installation-of-latest-generation-compact-driving-simulator-at-italian-braking-systems-manufacturer-brembo-1805/

[2] 2022. *What Is Simulated Annealing?* https://www.mathworks.com/help/gads/what-is-simulated-annealing.html

[3] 2023. . https://figshare.com/s/f07f53b2461e67418202

[4] 2023. *Adaptive Cruise Control for Passengers Cars.* https://www.bosch-mobility.com/en/solutions/assistance-systems/adaptive-cruise-control/

[5] 2023. *Simulink.* https://www.mathworks.com/products/simulink.html

[6] 2023. *Simulink Test.* https://www.mathworks.com/products/simulink-test.html

[7] Bestoun S Ahmed, Angelo Gargantini, and Miroslav Bures. 2020. An Automated Testing Framework For Smart TV apps Based on Model Separation. In *International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 62–73.

[8] Shaukat Ali, Lionel C Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. 2009. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering* 36, 6 (2009), 742–762.

[9] Matthias Althoff, Sebastian Maierhofer, and Christian Pek. 2020. Provably-correct and comfortable adaptive cruise control. *IEEE Transactions on Intelligent Vehicles* 6, 1 (2020), 159–174.

[10] Yashwanth Annpureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. 2011. S-TaLiRo: A Tool for Temporal Logic Falsification for Hybrid Systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, Parosh Aziz Abdulla and K. Rustan M. Leino (Eds.). Springer.

[11] Pier Giuseppe Anselma, Phillip J. Kollmeyer, Stefano Feraco, Angelo Bonfitto, Giovanni Belingardi, Ali Emadi, Nicola Amati, and Andrea Tonoli. 2021. Assessing Impact of Heavily Aged Batteries on Hybrid Electric Vehicle Fuel Economy and Drivability. In *IEEE Transportation Electrification Conference & Expo*.

[12] P Arcaini, E Riccobene, A Gargantini, et al. 2016. Model-based offline and online testing for medical software. In *European & Asian System, Software & Service Process Improvement & Innovation*. WHITEBOX, 11–20.

[13] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. 2011. Random testing: Theoretical results and practical implications. *Transactions on Software Engineering* 38, 2 (2011), 258–277.

[14] Aitor Arrieta, Shuai Wang, Goiuria Sagardui, and Leire Etxeberria. 2016. Test case prioritization of configurable cyber-physical systems with weight-based search algorithms. In *Genetic and Evolutionary Computation Conference*.

[15] Michele Asperti, Michele Vignati, and Edoardo Sabbioni. 2022. Driver-in-the-Loop Simulation to Assess Steering Torque Feeling due to Torque Vectoring Control. In *Vehicle Power and Propulsion Conference*. IEEE, 1–6.

[16] Mostafa Ayesh, Namya Mehan, Ethan Dhanraj, Abdul El-Rahwan, Simon Emil Opalka, Tony Fan, Akil Hamilton, Akshay Mathews Jacob, Rahul Anthony Sundarrajan, Bryan Widjaja, et al. 2022. Two Simulink Models with Requirements for a Simple Controller of a Pacemaker Device. In *International Workshop on Applied*, Vol. 90. 18–25.

[17] David Barnes, Jared Folden, Hwan-Sik Yoon, and Paulius Puzinauskas. 2020. *Scalable Simulation Environment for Adaptive Cruise Controller Development.* Technical Report. SAE Technical Paper.

[18] Daniel Goretti L Barroso, Ali Emadi, and Lucas Bruck. 2023. *Driver-in-the-Loop Drivability and Energy Efficiency Analysis of Regenerative Braking Strategies for Electric Vehicles.* Technical Report. SAE Technical Paper.

[19] Mirko Barthauer and Alexander Hafner. 2019. Testing an Adaptive Cruise Controller with coupled traffic and driving simulations.. In *SUMO*. 48–55.

[20] Stefan Björnander and Lars Grunske. 2008. Adaptive cruise controllers–a literature review. *Technical Rep. C4-01 TR M* 50 (2008).

[21] Alexander Boll, Florian Brokhausen, Tiago Amorim, Timo Kehrer, and Andreas Vogelsang. 2021. Characteristics, potentials, and limitations of open-source Simulink projects for empirical research. *Software and Systems Modeling* 20, 6 (2021), 2111–2130.

[22] Andrea Bombarda, Silvia Bonfanti, and Angelo Gargantini. 2022. Automatic Test Generation with ASMETA for the Mechanical Ventilator Milano Controller. In *Testing Software and Systems*. Springer, 65–72.

[23] Emanuele Bonera, Marco Gadola, Daniel Chindamo, Stefano Morbioli, and Paolo Magri. 2020. Integrated Design Tools for Model-Based Development of Innovative Vehicle Chassis and Powertrain Systems. In *Design Tools and Methods in Industrial Engineering*. Springer, 118–128.

[24] Lucas Bruck, Bruce Haycock, and Ali Emadi. 2021. A Review of Driving Simulation Technology and Applications. *IEEE Open Journal of Vehicular Technology* 2 (2021), 1–16. https://doi.org/10.1109/OJVT.2020.3036582

[25] Mattia Bruschetta, Enrico Picotti, Enrico Mion, Yutao Chen, Alessandro Beghi, and Diego Minen. 2019. A Nonlinear Model Predictive Control based Virtual Driver for high performance driving. In *Conference on Control Technology and Applications*. IEEE, 9–14.

[26] Alessandro Calò, Paolo Arcaini, Shaukat Ali, Florian Hauer, and Fuyuki Ishikawa. 2020. Simultaneously searching and solving multiple avoidable collisions for testing autonomous driving systems. In *Genetic and Evolutionary Computation Conference*. 1055–1063.

[27] Florent Colombet, Zhou Fang, and Andras Kemeny. 2017. Tilt thresholds for acceleration rendering in driving simulation. *Simulation* 93, 7 (2017), 595–603.

[28] Alexandre Donzé. 2010. Breach, A Toolbox for Verification and Parameter Synthesis of Hybrid Systems. In *Computer Aided Verification*. Springer, 167–170.

[29] Gidon Ernst, Paolo Arcaini, Georgios Fainekos, Federico Formica, Jun Inoue, Tanmay Khandait, Mohammad Mahdi Mahboob, Claudio Menghi, Giulia Pedrielli, Masaki Waga, Yoriyuki Yamagata, and Zhenya Zhang. 2022. ARCH-COMP 2022 Category Report: Falsification with Ubounded Resources. In *International Workshop on Applied Verification of Continuous and Hybrid Systems (EPiC Series in Computing, Vol. 90)*. EasyChair, 204–221.

[30] Gidon Ernst, Sean Sedwards, Zhenya Zhang, and Ichiro Hasuo. 2018. Fast Falsification of Hybrid Systems using Probabilistically Adaptive Input. In *International Conference on Quantitative Evaluation of Systems*. Springer, 165–181.

[31] Georgios E Fainekos and George J Pappas. 2006. Robustness of temporal logic specifications. In *Formal Approaches to Software Testing and Runtime Verification*. Springer, 178–192.

[32] Georgios E. Fainekos, Sriram Sankaranarayanan, Koichi Ueda, and Hakan Yazarel. 2012. Verification of automotive control applications using S-TaLiRo. In *2012 American Control Conference (ACC)*. 3567–3572. https://doi.org/10.1109/ACC.2012.6315384

[33] Alessandro Ferraris, Henrique de Carvalho Pinheiro, Edoardo Galanzino, Andrea Giancarlo Airale, and Massimiliana Carello. 2019. All-Wheel Drive Electric Vehicle Performance Optimization: From Modelling to Subjective Evaluation on a Static Simulator. In *Electric Vehicles International Conference*. 1–6. https://doi.org/10.1109/EV.2019.8893027

[34] Federico Formica, Mehrnoosh Askarpour, and Claudio Menghi. 2022. Search-based Software Testing Driven by Automatically Generated and Manually Defined Fitness Functions. arXiv preprint arXiv:2207.11016.

[35] Federico Formica, Tony Fan, Akshay Rajhans, Vera Pantelic, Mark Lawford, and Claudio Menghi. 2022. Simulation-based Testing of Simulink Models with Test Sequence and Test Assessment Blocks. https://arxiv.org/abs/2212.11589.

[36] Kamran Ghani, John A. Clark, and Yuan Zhan. 2009. Comparing algorithms for search-based test data generation of Matlab® Simulink® models. In *Congress on Evolutionary Computation*. IEEE, 2940–2947.

[37] Christoph Gladisch, Thomas Heinz, Christian Heinzemann, Jens Oehlerking, Anne von Vietinghoff, and Tim Pfitzer. 2019. Experience Paper: Search-Based Testing in Automated Driving Control Applications. In *International Conference on Automated Software Engineering*. IEEE/ACM.

[38] Liping Han, Tao Yue, Shaukat Ali, Aitor Arrieta, and Maite Arratibel. 2022. Are elevator software robust against uncertainties? results and experiences from an industrial case study. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 1331–1342.

[39] Mark Harman and Bryan F Jones. 2001. Search-based software engineering. *Information and software Technology* 43, 14 (2001), 833–839.

[40] Yizhuo He and Jie Pang. 2022. Optimization Design of Suspension Geometric Parameters of Formula Student Race Vehicle Based on ADAMS. In *International Conference on Manufacturing, Industrial Automation and Electronics*. IEEE, 164–169.

[41] Lawrence L Hoberock. 1976. *A survey of longitudinal acceleration comfort studies in ground transportation vehicles.* Technical Report. Council for Advanced Transportation Studies.

[42] Quanan Huang and Huiyi Wang. 2004. *Fundamental study of jerk: evaluation of shift quality and ride comfort.* Technical Report. SAE Technical Paper.

[43] Gunel Jahangirova, Andrea Stocco, and Paolo Tonella. 2021. Quality metrics and oracles for autonomous vehicles testing. In *Conference on Software Testing, Verification and Validation*. IEEE, 194–204.

[44] Monika Jaskolka, Vera Pantelic, Alan Wassyng, Mark Lawford, and Richard Paige. 2021. Repository Mining for Changes in Simulink Models. In *International Conference on Model Driven Engineering Languages and Systems*. ACM/IEEE, 46–57.

[45] Philip Koopman and Michael Wagner. 2016. Challenges in autonomous vehicle testing and validation. *SAE International Journal of Transportation Safety* 4, 1 (2016), 15–24.

[46] Markus Koschi, Christian Pek, Sebastian Maierhofer, and Matthias Althoff. 2019. Computationally efficient safety falsification of adaptive cruise control systems. In *Intelligent Transportation Systems Conference*. IEEE, 2879–2886.

[47] Thomas Laurent, Stefan Klikovits, Paolo Arcaini, Fuyuki Ishikawa, and Anthony Ventresque. 2022. Parameter Coverage for Testing of Autonomous Driving Systems Under Uncertainty. *ACM Transactions on Software Engineering and Methodology* (2022).

[48] Qin Lin, Sicco Verwer, and John Dolan. 2020. Safety verification of a data-driven adaptive cruise controller. In *Intelligent Vehicles Symposium*. IEEE, 2146–2151.

[49] Xiao Ling and Tim Menzies. 2023. On the Benefits of Semi-Supervised Test Case Generation for Cyber-Physical Systems. *arXiv preprint arXiv:2305.03714* (2023).

[50] Xiao Ling and Tim Menzies. 2023. What Not to Test (for Cyber-Physical Systems). *IEEE Transactions on Software Engineering* (2023), 1–17. https://doi.org/10.1109/TSE.2023.3272309

[51] Bing Liu, Shiva Nejati, and Lionel C Briand. 2019. Effective fault localization of automotive Simulink models: achieving the trade-off between test oracle effort and fault localization accuracy. *Empirical Software Engineering* 24 (2019), 444–490.

[52] Eduardo Louback, Fabricio Machado, Lucas Bruck, Phillip J. Kollmeyer, and Ali Emadi. 2022. Real-Time Performance and Driveability Analysis of a Clutchless Multi-Speed Gearbox for Battery Electric Vehicle Applications. In *IEEE Transportation Electrification Conference & Expo*.

[53] MARC 2022. *Adaptive Cruise Control System Market to Witness Strong Growth, with a Projected CAGR of 6.6% | Market.us Report.* http://www.globenewswire.com/news-release/2023/04/11/2644946/0/en/Adaptive-Cruise-Control-System-Market-to-Witness-Strong-Growth-with-a-Projected-CAGR-of-6-6-Market-us-Report.html

[54] MARC 2022. *Everything You Need to Know about Our Comprehensive Car Testing.* https://www.caranddriver.com/features/a32018270/how-we-test-cars/

[55] MARC 2022. *McMaster Automotive Resource Centre (MARC).* https://facilities.mcmaster.ca/building/mcmaster-automotive-resource-centre-marc/

[56] Anastasia Mavridou, Hamza Bourbouh, Dimitra Giannakopoulou, Thomas Pressburger, Mohammad Hejase, Pierre-Loic Garoche, and Johann Schumann. 2020. The ten lockheed martin cyber-physical challenges: formalized, analyzed, and explained. In *International Requirements Engineering Conference*. IEEE, 300–310.

[57] Aakar Mehra, Wen-Loong Ma, Forrest Berg, Paulo Tabuada, Jessy W Grizzle, and Aaron D Ames. 2015. Adaptive cruise control: Experimental validation of advanced controllers on scale-model cars. In *American control conference*. IEEE, 1411–1418.

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

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

[60] Thomas Müller, Hermann Hajek, Ljubica Radić-Weißenfeld, and Klaus Bengler. 2013. Can you feel the difference? The just noticeable difference of longitudinal acceleration. In *Human Factors and Ergonomics Society Annual Meeting*, Vol. 57. SAGE Publications Sage CA, 1219–1223.

[61] Daniel F Opila, Xiaoyong Wang, Ryan McGee, R Brent Gillespie, Jeffrey A Cook, and Jessy W Grizzle. 2011. An energy management controller to optimally trade off fuel economy and drivability for hybrid vehicles. *IEEE Transactions on Control Systems Technology* 20, 6 (2011), 1490–1505.

[62] Luigi Pariota, Angelo Coppola, Luca Di Costanzo, Claudio D'Aniello, and Gennaro Nicola Bifuclo. 2019. Motivating the need for an integrated software architecture for Connected and Automated Vehicles technologies development and testing. In *International Conference on Models and Technologies for Intelligent Transportation Systems*. 1–8.

[63] Jarkko Peltomäki and Ivan Porres. 2022. Falsification of Multiple Requirements for Cyber-Physical Systems Using Online Generative Adversarial Networks and Multi-Armed Bandits. In *International Conference on Software Testing, Verification and Validation Workshops*. IEEE.

[64] Herbert Schuette and Peter Waeltermann. 2005. Hardware-in-the-loop testing of vehicle dynamics controllers–a technical survey. *SAE transactions* (2005), 593–609.

[65] A Shaout and Mohammad Ameen Jarrah. 1997. Cruise control technology review. *Computers & electrical engineering* 23, 4 (1997), 259–271.

[66] Simscape 2022. *Objectified Evaluation at Engine-In-The-Loop Test Bench.* https://magazine.fev.com/en/drivability-characteristics/

[67] Andrea Stocco, Brian Pulfer, and Paolo Tonella. 2022. Mind the Gap! A Study on the Transferability of Virtual vs Physical-world Testing of Autonomous Driving Systems. *IEEE Transactions on Software Engineering* (2022), 1–13. https://doi.org/10.1109/TSE.2022.3202311

[68] Andrea Stocco, Brian Pulfer, and Paolo Tonella. 2023. Model vs system level testing of autonomous driving systems: a replication and extension study. *Empirical Software Engineering* 28, 3 (2023), 73.

[69] Taku Takahama and Daisuke Akasaka. 2018. Model predictive control approach to design practical adaptive cruise control for traffic jam. *International journal of automotive engineering* 9, 3 (2018), 99–104.

[70] TestAssessment 2022. *Test Assessment.* https://www.mathworks.com/help/sltest/ref/testassessment.html

[71] TestSequence 2022. *Test Sequence.* https://www.mathworks.com/help/sltest/ref/testsequence.html

[72] Quinn Thibeault, Jacob Anderson, Aniruddh Chandratre, Giulia Pedrielli, and Georgios Fainekos. 2021. PSY-TaLiRo: A Python Toolbox for Search-Based Test Generation for Cyber-Physical Systems. In *Formal Methods for Industrial Critical Systems*, Alberto Lluch Lafuente and Anastasia Mavridou (Eds.). Springer, 223–231.

[73] Eugenio Tramacere, Luis Miguel Molina Castellanos, Nicola Amati, Andrea Tonoli, and Angelo Bonfitto. 2022. Adaptive LQR Control for a Rear-Wheel Steering Battery Electric Vehicle. In *Vehicle Power and Propulsion Conference*. IEEE.

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

[75] Cumhur Erkan Tuncali, Theodore P. Pavlic, and Georgios Fainekos. 2016. Utilizing S-TaLiRo as an automatic test generation framework for autonomous vehicles. In *International Conference on Intelligent Transportation Systems*. IEEE, 1470–1475.

[76] Pablo Valle, Aitor Arrieta, and Maite Arratibel. 2023. Automated Misconfiguration Repair of Configurable Cyber-Physical Systems with Search: an Industrial Case Study on Elevator Dispatching Algorithms. In *International Conference on Software Engineering: Software Engineering in Practice*. IEEE.

[77] VI-CarRealTime-Simulink 2022. *VI-CarRealTime.* https://www.vi-grade.com/en/products/vi-carrealtime/

[78] Masaki Waga. 2020. Falsification of Cyber-Physical Systems with Robustness-Guided Black-Box Checking. In *International Conference on Hybrid Systems: Computation and Control*. ACM, Article 11, 13 pages.

[79] Qinghua Xu, Shaukat Ali, Tao Yue, and Maite Arratibel. 2022. Uncertainty-aware transfer learning to evolve digital twins for industrial elevators. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 1257–1268.

[80] Yoriyuki Yamagata, Shuang Liu, Takumi Akazaki, Yihai Duan, and Jianye Hao. 2021. Falsification of cyber-physical systems using deep reinforcement learning. *IEEE Transactions on Software Engineering* 47, 12 (2021), 2823–2840. https://doi.org/10.1109/TSE.2020.2969178

[81] Zhenya Zhang, Deyun Lyu, Paolo Arcaini, Lei Ma, Ichiro Hasuo, and Jianjun Zhao. 2021. Effective Hybrid System Falsification Using Monte Carlo Tree Search Guided by QB-Robustness. In *Computer Aided Verification*. Springer, 1–24.