Search-Based Automated Testing of Continuous Controllers: Framework, Tool Support, and Case Studies

Reza Matinnejad, Shiva Nejati, Lionel Briand

SnT center, University of Luxembourg, 4 rue Alphonse Weicker, L-2721 Luxembourg, Luxembourg

Thomas Bruckmann, Claude Poull

Delphi Automotive Systems, Avenue de Luxembourg, L-4940 Bascharage, Luxembourg

Abstract

Context. Testing and verification of automotive embedded software is a major challenge. Software production in automotive domain comprises three stages: Developing automotive functions as Simulink models, generating code from the models, and deploying the resulting code on hardware devices. Automotive software artifacts are subject to three rounds of testing corresponding to the three production stages: Model-in-the-Loop (MiL), Software-in-the-Loop (SiL) and Hardware-in-the-Loop (HiL) testing.

Objective. We study testing of continuous controllers at the Model-in-Loop (MiL) level where both the controller and the environment are represented by models and connected in a closed loop system. These controllers make up a large part of automotive functions, and monitor and control the operating conditions of physical devices.

Method. We identify a set of requirements characterizing the behavior of continuous controllers, and develop a search-based technique based on random search, adaptive random search, hill climbing and simulated annealing algorithms to automatically identify worst-case test scenarios which are utilized to generate test cases for these requirements.

Results. We evaluated our approach by applying it to an industrial automotive controller (with 443 Simulink blocks) and to a publicly available controller (with 21 Simulink blocks). Our experience shows that automatically generated test cases lead to MiL level simulations indicating potential violations of the system requirements. Further, not only does our approach generate significantly better test cases faster than random test case generation, but it also achieves better results than test scenarios devised by domain experts. Finally, our generated test cases uncover discrepancies between environment models and the real world when they are applied at the Hardware-in-the-Loop

Email addresses: reza.matinnejad@uni.lu (Reza Matinnejad), shiva.nejati@uni.lu (Shiva Nejati), lionel.briand@uni.lu (Lionel Briand), thomas.bruckmann@delphi.com (Thomas Bruckmann), claude.poull@delphi.com (Claude Poull)

(HiL) level.

Conclusion. We propose an automated approach to MiL testing of continuous controllers using search. The approach is implemented in a tool and has been successfully applied to a real case study from the automotive domain.

Keywords: Search-Based Testing, Continuous Controllers, Model-in-the-Loop Testing, Automotive Software Systems, Simulink Models

---

1. Introduction

Modern vehicles are increasingly equipped with Electronic Control Units (ECUs). The amount and the complexity of software embedded in the ECUs of today's vehicles is rapidly increasing. To ensure the high quality of software and software-based functions on ECUs, the automotive and ECU manufacturers have to rely on effective techniques for verification and validation of their software systems. A large group of automotive software functions require to monitor and control the operating conditions of physical components. Examples are functions controlling engines, brakes, seatbelts, and airbags. These controllers are widely studied in the control theory domain as *continuous controllers* [1, 2] where the focus has been to *optimize* their design for a specific application or a specific hardware configuration [3, 4, 5]. Yet a complementary and important problem, of how to systematically and automatically *test* controllers to ensure their correctness and safety, has received almost no attention in control engineering research [1].

In this article, we concentrate on the problem of automatic and systematic test case generation for continuous controllers. The principal challenges when analyzing such controllers stem from their continuous interactions with the physical environment, usually through feedback loops where the environment impacts computations and vice versa. We study the testing of controllers at an early stage where both the controller and the environment are represented by models and connected in a closed feedback loop process. In model-based approaches to embedded software design, this level is referred to as *Model-in-the-Loop (MiL)* testing.

Testing continuous aspects of control systems is challenging and is not yet supported by existing tools and techniques [1, 3, 4]. There is a large body of research on testing *mixed* discrete-continuous behaviors of embedded software systems where the system under test is represented using state machines, hybrid automata, and hybrid petri nets [6, 7, 8]. For these models, test case generation techniques have been introduced based on meta-heuristic search and model checking [9, 10]. These techniques, however, are not amenable to testing *purely* continuous properties of controllers described in terms of mathematical models, and in particular, differential equations. A number of commercial verification and testing tools have been developed, aiming to generate test cases for MATLAB/Simulink models, namely the Simulink Design Verifier software [11], and Reactis Tester [12]. Currently, these tools handle only combinatorial and logical blocks of the MATLAB/Simulink models, and fail to generate test cases that specifically evaluate continuous blocks (e.g., integrators) [1].

Contributions. In this article, we propose a search-based approach to automate generation of MiL level test cases for continuous controllers. We identify a set of common

requirements characterizing the desired behavior of such controllers. We develop a search-based technique to generate stress test cases attempting to violate these requirements by combining *explorative* and *exploitative* search algorithms [13]. Specifically, we first apply a purely explorative random search to evaluate a number of input signals distributed across the search space. Combining the domain experts' knowledge and random search results, we select a number of regions that are more likely to lead to critical requirement violations in practice. We then start from the worst-case input signals found during exploration, and apply an exploitative *single-state* search [13] to the selected regions to identify test cases for the controller requirements. Our search algorithms rely on *objective* functions created by formalizing the controller requirements.

We have implemented our approach in a tool, called Continuous Controller Tester (CoCoTest). We evaluated our approach by applying it to an automotive air compressor module and to a publicly available controller model. Our experiments show that our approach automatically generates several test cases for which the MiL level simulations indicate potential errors in the controller model or in the environment model. Furthermore, the resulting test cases had not been previously found by manual testing based on domain expertise. In addition, our approach computes test cases better and faster than a random test case generation strategy. Finally, our generated test cases uncover discrepancies between environment models and the real world when they are applied at the Hardware-in-the-Loop (HiL) level.

Organization. This article is organized as follows. Section 2 introduces the industrial background of our research. Section 3 precisely formulates the problem we aim to address in this article. Section 4 outlines our solution approach and describes how we cast our MiL testing approach as a search problem. Our MiL testing tool, CoCoTest, and the results of our evaluation of the proposed MiL testing approach are presented in Sections 5 and 6, respectively. Section 7 compares our contributions with related work. Finally, Section 8 concludes the article.

2. MiL Testing of Continuous Controllers: Practice and Challenges

Control system development involves building of control software (controllers) to interact with mechatronic systems usually referred to as plants or environment [2]. An abstract view of such controllers and their plant models is shown in Figure 1(a). These controllers are commonly used in many domains such as manufacturing, robotics, and automotive. Model-based development of control systems is typically carried out in three major levels described below. The models created through these levels become increasingly more similar to real controllers, while verification and testing of these models becomes successively more complex and expensive.

Model-in-the-Loop (MiL): At this level, a model for the controller and a model for the plant are created in the same notation and connected in the same diagram. In many sectors and in particular in the automotive domain, these models are created in MATLAB/Simulink. The MiL simulation and testing is performed entirely in a virtual environment and without any need for any physical component. The focus of MiL testing is to verify the control behavior or logic, and to ensure that the interactions between the controller and the plant do not violate the system requirements.

Software-in-the-Loop (SiL): At this level, the controller model is converted to code (either autocoded or manually). This often includes the conversion of floating point

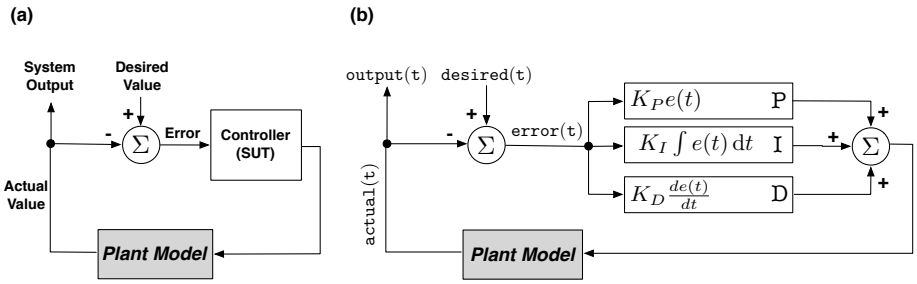

Figure 1: Continuous controllers: (a) A MiL level controller-plant model, and (b) a generic PID formulation of a continuous controller.

data types into fixed-point values as well as addition of hardware-specific libraries. The testing at the SiL level is still performed in a virtual and simulated environment like MiL, but the focus is on controller code which can run on the target platform. Further, in contrast to verifying the behavior, SiL testing aims to ensure correctness of floating point to fixed-point conversion and the conformance of code to control models, especially in contexts where coding is (partly) manual.

Hardware-in-the-Loop (HiL): At this level, the controller software is fully installed into the final control system (e.g., in our case, the controller software is installed on the ECUs). The plant is either a real piece of hardware, or is some software (HiL plant model) that runs on a real-time computer with physical signal simulation to lead the controller into believing that it is being used on a real plant. The main objective of HiL is to verify the integration of hardware and software in a more realistic environment. HiL testing is the closest to reality, but is also the most expensive among the three testing levels and performing the test takes the longest at this level.

In this article, among the above three levels, we focus on the MiL level testing. MiL testing is the primary level intended for verification of the control behavior and ensuring the satisfaction of their requirements. Development and testing at this level is considerably fast as the engineers can quickly modify the control model and immediately test the system. Furthermore, MiL testing is entirely performed in a virtual environment, enabling execution of a large number of test cases. Finally, the MiL level test cases can later be used at SiL and HiL levels either directly or after some adaptations.

Currently, in most companies, MiL level testing of controllers is limited to running the controller-plant Simulink model (e.g., Figure 1(a)) for a small number of simulations, and manually inspecting the results of individual simulations. The simulations are often selected based on the engineers' domain knowledge and experience, but in a rather ad hoc way. Such simulations are useful for checking the overall sanity of the control behavior, but cannot be taken as a substitute for systematic MiL testing. Manual simulations fail to find erroneous scenarios that the engineers might not be aware of a priori. Identifying such scenarios later during SiL/HiL is much more difficult and expensive than during MiL testing. Also, manual simulation is by definition limited in scale and scope.

Our goal is to develop an automated MiL testing technique to verify controller-plant

systems. To do so, we formalize the properties of continuous controllers regarding the functional, performance, and safety aspects. We develop an automated test case generation approach to evaluate controllers with respect to these properties. In our work, the test inputs are signals, and the test outputs are measured over the simulation diagrams generated by MATLAB/Simulink plant models. The simulations are discretized where the controller output is sampled at a rate of a few milliseconds. To generate test cases, we combine two search algorithms: (1) An explorative random search that allows us to achieve high diversity of test inputs in the space, and to identify the most critical regions that need to be explored further. (2) An exploitative search that enables us to focus our search and compute worst-case scenarios in the identified critical regions.

3. Problem Formulation

Figure 1(a) shows an overview of a controller-plant model at the MiL level. Both the controller and the plant are captured as models and linked via virtual connections. We refer to the input of the controller-plant system as *desired* or *reference* value. For example, the desired value may represent the location we want a robot to move to, the speed we require an engine to reach, or the position we need a valve to arrive at. The *system output* or the *actual* value represents the actual state/position/speed of the hardware components in the plant. The actual value is expected to reach the desired value over a certain time limit, making the *Error*, i.e., the difference between the actual and desired values, eventually zero. The task of the controller is to eliminate the error by manipulating the plant to obtain the desired effect on the system output.

The overall objective of the controller in Figure 1(a) may sound simple. In reality, however, the design of such controllers requires calculating proper corrective actions for the controller to stabilize the system within a certain time limit, and further, to guarantee that the hardware components will eventually reach the desired state without oscillating too much around it and without any damage. A controller design is typically implemented via complex differential equations known as *proportional-integral-derivative (PID)* [2]. Figure 1(b) shows the generic (most basic) formulation of a PID equation. Let $e(t)$ be the difference between $desired(t)$ and $actual(t)$ (i.e., error). A PID equation is a summation of three terms: (1) a proportional term $K_P e(t)$, (2) an integral term $K_I \int e(t)dt$, and (3) a derivative term $K_D \frac{de(t)}{dt}$. Note that the PID formulation for real world controllers are more complex than the formula shown in Figure 1(b). Figure 2 shows a typical output diagram of a PID controller. As shown in the figure, the actual value starts at an initial value (here zero), and gradually moves to reach and stabilize at a value close to the desired value.

Continuous controllers are characterized by a number of generic requirements discussed in Section 3.1. Having specified the requirements of continuous controllers, we show in Section 3.2 how we define testing objectives based on these requirements, and how we formulate MiL testing of continuous controllers as a search problem.

3.1. Testing Continuous Controller Requirements

To ensure that a controller design is satisfactory, engineers perform several simulations, and analyze the output simulation diagram (Figure 2) with respect to a number

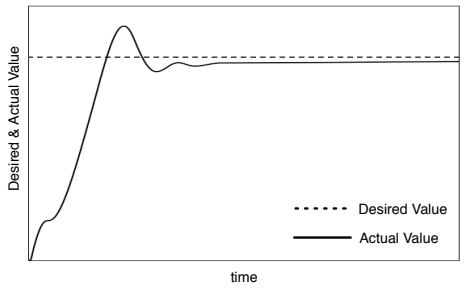

Figure 2: A typical example of a continuous controller output.

of requirements. After careful investigations, we identified the following requirements for controllers:

Liveness (functional): The controller shall guarantee that the actual value will reach the desired value within t_1 seconds. This is to ensure that the controller indeed satisfies its main functional requirement.

Stability (safety, functional): The actual value shall stabilize at the desired value after t_2 seconds. This is to make sure that the actual value does not divert from the desired value or does not keep oscillating around the desired value after reaching it.

Smoothness (safety): The actual value shall not change abruptly when it is close to the desired one. That is, the difference between the actual and desired values shall not exceed v_2, once the difference has already reached v_1 for the first time. This is to ensure that the controller does not damage any physical devices by sharply changing the actual value when the error is small.

Responsiveness (performance): The difference between the actual and desired values shall be at most v_3 within t_3 seconds, ensuring the controller responds within a time limit.

The above four requirement templates are illustrated on a typical controller output diagram in Figure 3 where the parameters t_1, t_2, t_3, v_1, v_2, and v_3 are represented. The first three parameters represent time while the last three are described in terms of the controller output values. As shown in the figure, given specific controller requirements with concrete parameters and given an output diagram of a controller under test, we can determine whether that particular controller output satisfies the given requirements.

Having discussed the controller requirements and outputs, we now describe how we generate input test values for a given controller. Typically, controllers have a large number of configuration parameters that affect their behaviors. For the configuration parameters, we use a value assignment commonly used for HiL testing because it enables us to compare the results of MiL and HiL testing. In our approach, we focus on two essential controller inputs in our MiL testing approach: (1) the initial actual value, and (2) the desired value. Among these two inputs, the desired value can be easily manipulated externally. However, since the controller is a closed loop system, it is not generally possible to modify the initial actual value and start the system from an arbitrary initial state. In general, the initial actual state, which is usually set to zero,

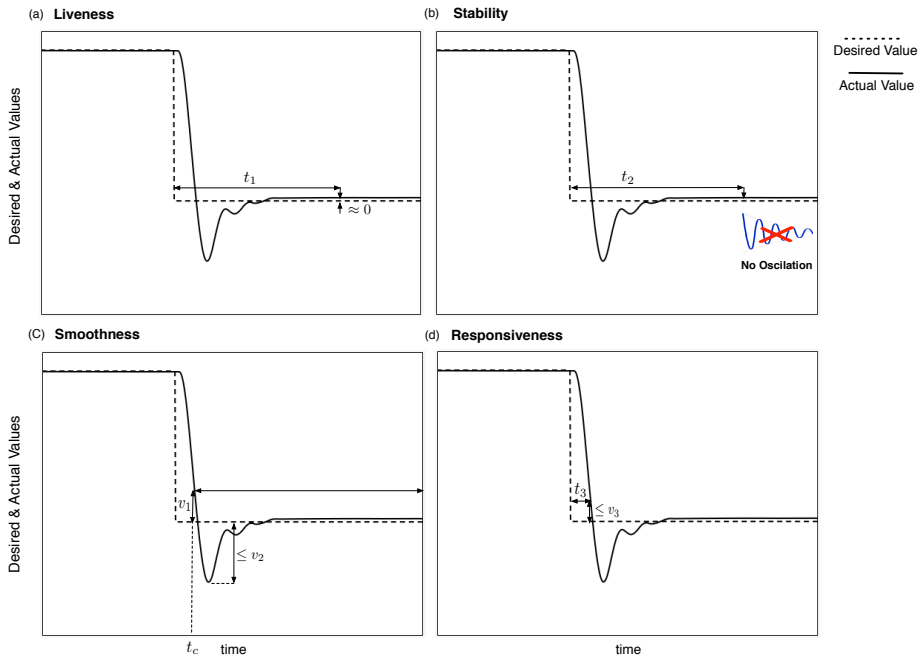

Figure 3: The controller requirements illustrated on the controller output: (a) Liveness, (b) Stability, (c) Smoothness, and (d) Responsiveness.

depends on the plant model and cannot be manipulated externally. Assuming that the system always starts from zero is like testing a cruise controller only for positive car speed increases, and missing a whole range of speed decreasing scenarios.

To eliminate this restriction, we provide a step signal for the desired value of the controller (see examples of step signals in Figure 3 and Figure 4(a)). The step signal consists of two consecutive constant signals such that the first one sets the controller at the *initial* desired value, and the second one moves the controller to the *final* desired value. The lengths of the two signals in a step signal are equal (see Figure 4(a)), and should be sufficiently long to give the controller enough time to stabilize at each of the initial and final desired values. Figure 4(b) shows an example of a controller output diagram for the input step signal in Figure 4(a).

3.2. Formulating MiL Testing as a Search Problem

Given a controller-plant model and a set of controller requirements, the goal of MiL testing is to generate controller input values such that the resulting controller output values violate, or become close to violating, the given requirements. Based on this description, any MiL testing strategy has to perform the following common tasks: (1) It should generate input signals to the controller, i.e., step signal in Figure 4(a). (2) It should receive the output, i.e., Actual in Figure 4(b), from the controller model, and evaluate the output against the controller requirements. Below, we first formalize the controller input and output, and then, we derive five objective functions from the four requirements introduced in Section 3.1. Specifically, we develop one objective function

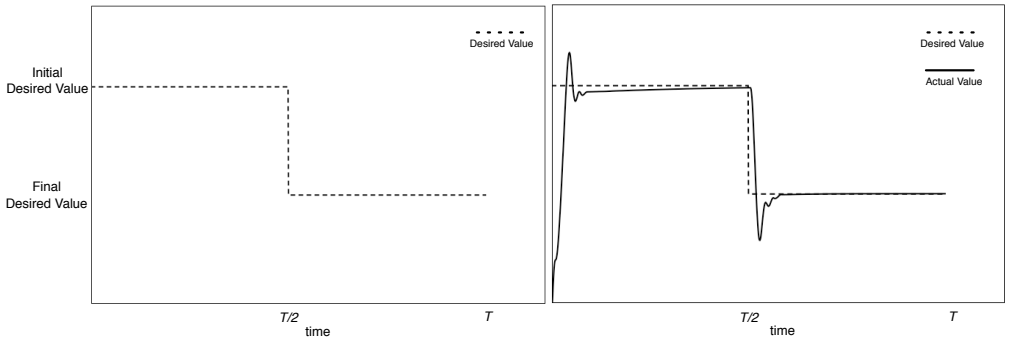

Figure 4: Controller input step signals: (a) Step signal. (b) Output of the controller (actual) given the input step signal (desired).

for each of the liveness, stability and responsiveness requirements, and two objective functions for the smoothness requirement.

Controller input and output: Let $\mathcal{T} = \{0, \ldots, T\}$ be a set of time points during which we observe the controller behavior, and let `min` and `max` be the minimum and maximum values for the Actual and Desired attributes in Figure 1(a). In our work, since input is assumed to be a step signal, the observation time T is chosen to be long enough so that the controller can reach and stabilize at two different Desired positions successively (e.g., see Figure 4(b)). Note that Actual and Desired are of the same type and bounded within the same range, denoted by $[\texttt{min} \ldots \texttt{max}]$. As discussed in Section 3, the test inputs are step signals representing the Desired values (e.g. Figure 4(a)). We define an input step signal in our approach to be a function $\texttt{Desired} : \mathcal{T} \rightarrow \{\texttt{min}, \ldots, \texttt{max}\}$ such that there exists a pair `Initial Desired` and `Final Desired` of values in $[\texttt{min} \ldots \texttt{max}]$ that satisfy the following conditions:

$\forall t \cdot 0 \leq t < \frac{T}{2} \Rightarrow \texttt{Desired}(t) = \texttt{Initial Desired} \bigwedge$

$\forall t \cdot \frac{T}{2} \leq t < T \Rightarrow \texttt{Desired}(t) = \texttt{Final Desired}$

We define the controller output, i.e., Actual, to be a function $\texttt{Actual} : \mathcal{T} \rightarrow \{\texttt{min}, \ldots, \texttt{max}\}$ that is produced by the given controller-plant model, e.g., in MATLAB/Simulink environment.

The search space in our problem is the set of all possible input functions, i.e., the `Desired` function. Each `Desired` function is characterized by the pair `Initial Desired` and `Final Desired` values. In control system development, it is common to use floating point data types at MiL level. Therefore, the search space in our work is the set of all pairs of floating point values for `Initial Desired` and `Final Desired` within the $[\texttt{min} \ldots \texttt{max}]$ range.

Objective Functions: Our goal is to guide the search to identify input functions in the search space that are more likely to break the properties discussed in Section 3.1. To do so, we create the following five objective functions:

- **Liveness:** Let t_1 be the liveness property parameter in Section 3.1. We define the liveness objective function O_L as:

  $$max_{t_1 + \frac{T}{2} < t \leq T} \{|\texttt{Desired}(t) - \texttt{Actual}(t)|\}$$

  That is, O_L is the maximum of the difference between `Desired` and `Actual` after time $t_1 + \frac{T}{2}$.

- **Stability:** Let t_2 be the stability property parameter in Section 3.1. We define

the stability objective function O_{St} as:

$$StdDev_{t_2+\frac{T}{2}<t\leq T}\{\texttt{Actual}(t)\}$$

That is, O_{St} is the standard deviation of the values of `Actual` function between $t_2 + \frac{T}{2}$ and T.

- **Smoothness**: Let v_1 be the smoothness property parameter in Section 3.1. Let $tc \in \mathcal{T}$ be such that $tc > \frac{T}{2}$ and

$$|\texttt{Desired}(tc) - \texttt{Actual}(tc)| \leq v_1 \ \bigwedge$$
$$\forall t \cdot \tfrac{T}{2} \leq t < tc \Rightarrow |\texttt{Desired}(t) - \texttt{Actual}(t)| > v_1$$

  That is, tc is the first point in time after $\frac{T}{2}$ where the difference between `Actual` and `Final Desired` values has reached v_1. We then define the smoothness objective function O_{Sm} as:

$$max_{tc<t\leq T}\{|\texttt{Desired}(t) - \texttt{Actual}(t)|\}$$

  That is, the function O_{Sm} is the maximum difference between `Desired` and `Actual` after tc.

  Note that O_{Sm} measures the *absolute* value of undershoot/overshoot. We noticed in our work that in addition to finding the largest undershoot/overshoot scenarios, we need to identify scenarios where the overshoot/undershoot is large compared to the step size in the input step signal. In other words, we are interested in scenarios where a small change in the position of the controller, i.e., a small difference between `Initial Desired` and `Final Desired`, yields a *relatively* large overshoot/undershoot. These latter scenarios cannot be identified if we only use the O_{Sm} function for evaluating the Smoothness requirement. Therefore, we define the next objective function, normalized smoothness, to help find such test scenarios.

- **Normalized Smoothness**: We define the normalized smoothness objective function O_{Ns}, by normalizing the O_{Sm} function:

$$O_{Ns} = \frac{O_{Sm}}{O_{Sm} + |\texttt{Final Desired} - \texttt{Initial Desired}|}$$

  O_{Ns} evaluates the overshoot/undershoot values relative to the step size of the input step signal.

- **Responsiveness**: Let v_3 be the responsiveness parameter in Section 3.1. We define the responsiveness objective function O_R to be equal to tr such that $tr \in \mathcal{T}$ and $tr > \frac{T}{2}$ and

$$|\texttt{Desired}(tr) - \texttt{Actual}(tr)| \leq v_3 \ \bigwedge$$
$$\forall t \cdot \tfrac{T}{2} \leq t < tr \Rightarrow |\texttt{Desired}(t) - \texttt{Actual}(t)| > v_3$$

  That is, O_R is the first point in time after $\frac{T}{2}$ where the difference between `Actual` and `Final Desired` values has reached v_3.

We did not use v_2 from the smoothness and t_3 from the responsiveness properties in definitions of O_{Sm} and O_R. These parameters determine pass/fail conditions for test cases, and are not required to guide the search. Further, v_2 and t_3 depend on the specific hardware characteristics and vary from customer to customer. Hence, they are

not known at the MiL level. Specifically, we define O_{Sm} to measure the maximum overshoot rather than to determine whether an overshoot exceeds v_2, or not. Similarly, we define O_R to measure the actual response time without comparing it with t_3.

The above five objective functions are heuristics and provide quantitative estimates of the controller requirements, allowing us to compare different test inputs. The higher the objective function value, the more likely it is that the test input violates the requirement corresponding to that objective function. We use these objective functions in our search algorithms discussed in the next section.

In the next section, we describe our automated search-based approach to MiL testing of controller-plant systems. Our approach automatically generates input step signals such as the one in Figure 4(a), produces controller output diagrams for each input signal, and evaluates the five controller objective functions on the output diagram. Our search is guided by a number of heuristics to identify the input signals that are more likely to violate the controller requirements. Our approach relies on the fact that, during MiL, a large number of simulations can be generated quickly and without breaking any physical devices. Using this characteristic, we propose to replace the existing manual MiL testing with our search-based automated approach that enables the evaluation of a large number of output simulation diagrams and the identification of critical controller input values.

4. Solution Approach

In this section, we describe our search-based approach to MiL testing of controllers, and show how we employ and combine different search algorithms to guide and automate MiL testing of continuous controllers. Figure 5 shows an overview of our automated MiL testing approach. In the first step, we receive a controller-plant model (e.g., in MATLAB/Simulink) and a set of objective functions derived from requirements. We partition the input search space into a set of regions, and for each each region, compute a value indicating the evaluation of a given objective function on that region based on random search (exploration). We refer to the result as a *HeatMap* diagram [14]. HeatMap diagrams are graphical 2-D or 3-D representations of data where a matrix of values are represented by colors. In this paper, we use grayscale 2-D HeatMap diagrams (see Figure 6(b) for an example). These diagrams are intuitive and easy to understand by the users of our approach. In addition, our HeatMap diagrams are divided into equal regions (squares), making it easier for engineers to delineate critical parts of the input space in terms of these equal and regular-shape regions. Based on domain expert knowledge, we select some of the regions that are more likely to include critical and realistic errors. In the second step, we focus our search on the selected regions and employ a single-state heuristic search to identify, within those regions, the worst-case scenarios to test the controller. Single-state search optimizers only keep one candidate solution at a time, as opposed to *population-based* algorithms that maintain a set of samples at each iteration [13].

In the first step of our approach in Figure 5, we apply a random (unguided) search to the entire search space in order to identify high risk areas. The search explores diverse test inputs to provide an unbiased estimate of the average objective function values in different regions of the search space. In the second step, we apply a heuristic

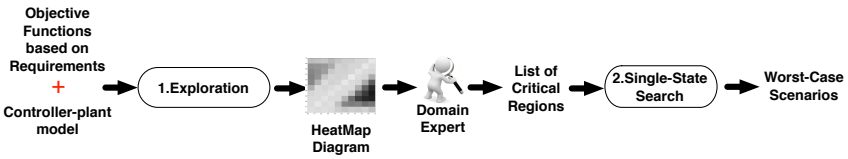

Figure 5: An overview of our automated approach to MiL testing of continuous controllers.

single-state search to a selection of regions in order to find worst-case test scenarios that are likely to violate the controller properties. In the following two sections, we discuss these two steps.

4.1. Exploration

Figure 6(a) shows the exploration algorithm used in the first step. The algorithm takes as input a controller-plant model M and an objective function O, and produces a HeatMap diagram (e.g., see Figure 6(b)). Briefly, the algorithm divides the input search space S of M into a number of equal regions. It then generates a random point p in S in line 4. The dimensions of p characterize an input step function `Desired` which is given to M as input in line 6. The model M is executed in Matlab/Simulink to generate the `Actual` output. The objective function O is then computed based on the `Desired` and `Actual` functions. The tuple (p, o) where o is the value of the objective function at p is added to P. The algorithm stops when the number of generated points in each region is at least N. Finding an appropriate value for N is a trade off between accuracy and efficiency. Since executing M is relatively expensive, it is often not practical to generate many points (large N). Likewise, a small number of points in each region is unlikely to give us an accurate estimate of the average objective function for that region.

Input: The exploration algorithm in Figure 6(a) takes as input a controller-plant model M, an objective function O, and an observation time T. Note that the controller model is an abstraction of the software under test, and the plant model is required to simulate the controller model, and evaluate the objective functions. The length of simulation is determined by the observation time T. Finally, the objective function O is chosen among the five objective functions described in section 3.2.

Output: The output of the algorithm in Figure 6(a) is a set P of (p, o) tuples where p is a point in the search space and o is the objective function value for p. The set P is visualized as a HeatMap diagram [14] where the axes are the initial and final desired values. In HeatMaps, each region is assigned the average value of the values of the points within that region. The intervals of the region values are then mapped into different shades, generating a shaded diagram such as the one in Figure 6(b). In our work, we generate five HeatMap diagrams corresponding to the five objective functions O_L, O_{St}, O_{Sm}, O_{Ns} and O_R discussed in Section 3.2. The HeatMap diagrams generated in the first step are reviewed by domain experts. They select a set of regions that are more likely to include realistic and critical inputs. For example, the diagram in Figure 6(b) is generated based on an air compressor controller model evaluated for the smoothness objective function O_{Sm}. This controller compresses the air by moving a flap between its open position (indicated by 0) and its closed position (indicated by 1.0). There are about 10 to 12 dark regions, i.e., the regions with the highest O_S values in Figure 6(b).

(a)

Algorithm. EXPLORATION

Input: A controller-plant model M with input search space S.
An objective function O.
An observation time T.

Output: An overview diagram (HeatMap) HM.
The set P of HM points.

1. Partition S into equal sub-regions
2. Let $P = \{\}$
3. **repeat**
4. Let $p = (\texttt{Initial Desired}, \texttt{Final Desired})$ be a random point in S
5. Let `Desired` be a step function generated based on p and T
6. Run M with the `Desired` input to obtain the `Actual` output
7. $o = O(\texttt{Desired}, \texttt{Actual})$
8. $P = \{(p, o)\} \cup P$
9. **until** there are at least N points in each region of S **do**
10. Create a HeatMap diagram HM based on P
11. Return HM and P

(b)

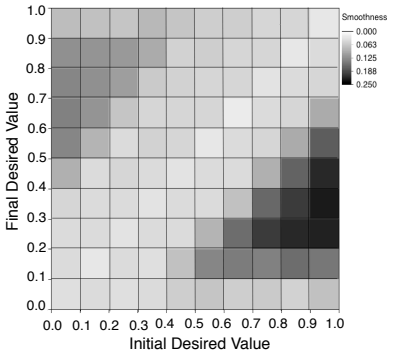

Figure 6: The first step of our approach in Figure 5: (a) The exploration algorithm. (b) An example HeatMap diagram produced by the algorithm in (a)

These regions have initial flap positions between 0.5 and 1.0, and final flap positions between 0.1 and 0.6. Among these regions, domain experts tend to focus on regions with initial values between 0.8 and 1.0, or final values between 0.8 and 1.0. This is because, in practice, there is more probability of damage when a closed (or a nearly closed) flap is being moved, or when a flap is about to be closed.

Search heuristics: Figure 6(a) shows a generic description of our exploration algorithm. This algorithm can be implemented in different ways by devising various heuristics for generating random points in the input search space S (line 4 in Figure 6(a)). In our work, we use two alternative heuristics for line 4 of the algorithm: (1) a simple random search, which we call naive random search, and (2) an adaptive random search algorithm [13]. The naive random search simply calls a random function to generate points in line 4 in Figure 6(a). Adaptive random search is an extension of the naive random search that attempts to maximize the euclidean distance between the selected points. Specifically, it explores the space by iteratively selecting points in areas of the space where fewer points have already been selected. To implement adaptive random search, the algorithm in Figure 6(a) changes as follows: Let P_i be the set of points selected by adaptive random search at iteration i (line 8 in Figure 6(a)). At iteration $i+1$, at line 4, instead of generating one point, adaptive random search randomly generates a set X of candidate points in the input space. The search computes distances between each candidate point $p \in X$ and the points in P_i. Formally, for each point $p = (v_1, v_2)$ in X, the search computes a function $dist(p)$ as follows:

$$dist(p) = \text{MIN}_{(v'_1, v'_2) \in P_i} \sqrt{(v_1 - v'_1)^2 + (v_2 - v'_2)^2}$$

The search algorithm then picks a point $p \in X$ such that $dist(p)$ is the largest, and proceeds to the lines 5 to 7. Finally, the selected p together with the value o of objective function at point p is added to P_i to generate P_{i+1} in line 8.

The algorithm in Figure 6(a) stops when at least N points have been selected in each region. We anticipate the adaptive random heuristic reaches this termination condition faster than a naive random heuristic because it generates points that are more

evenly distributed across the entire space. Our work is similar to *quasi-random number generators* that are available in some languages, e.g., MATLAB [15]. Similar to our adaptive random search algorithm, these number generators attempt to minimize the discrepancy between the distribution of generated points. We evaluate efficiency of naive random search and adaptive random search in generating HeatMap diagrams in Section 6.

Note that a simple and faster solution to build HeatMap diagrams could have been to simply generate equally spaced points in the search space. However, assuming the software under test might have some regularity in its behavior, such a strategy might make it impossible to collect a statistically, unbiased representative subset of observations in each region.

4.2. Single-State Search

Figure 7(a) presents our single-state search algorithm for the second step of the procedure in Figure 5. The single-state search algorithm starts with the point with the worst (highest) objective function value among those computed by the random search in Figure 6(a). It then iteratively generates new points by tweaking the current point (line 6) and evaluates the given objective function on the newly generated points. Finally, it reports the point with the worst (highest) objective function value. In contrast to random search, the single-state search is guided by an objective function and performs a tweak operation. Since the search is driven by the objective function, we have to run the search five times separately for O_L, O_{St}, O_{Sm}, O_{Ns} and O_R.

At this step, we rely on single-state exploitative algorithms. These algorithms mostly make local improvements at each iteration, aiming to find and exploit local gradients in the space. In contrast, explorative algorithms, such as the adaptive random search we used in the first step, mostly wander about randomly and make big jumps in the space to explore it. We apply explorative search at the beginning to the entire search space, and then focus on a selected area and try to find worst case scenarios in that area using exploitative algorithms.

Input: The input to the single-state search algorithm in Figure 7(a) is the controller-plant model M, an objective function O, an observation time T, a HeatMap region r, and the set P of points generated by exploration algorithm in Figure 6(a). We already explained the input values M, O and T in Section 4.1. Region r is chosen among the critical regions of the HeatMap identified by the domain expert. The single-state search algorithm focuses on the input region r to find a worst-case test scenario of the controller. The algorithm, further, requires P to identify its starting point in r, i.e., the point with the worst (highest) objective function in r.

Output: The output of the single-state search algorithm is a worst-case test scenario found by the search after K iterations. For instance Figure 7(b) shows the worst-case scenario computed by our algorithm for the smoothness objective function applied to an air compressor controller. As shown in the figure, the controller has an undershoot around 0.2 when it moves from an initial desired value of 0.8 and is about to stabilize at a final desired value of 0.3.

Search heuristics: In our work, evaluating fitness functions takes a relatively long

(a)

Algorithm. SINGLESTATESEARCH
Input: A controller-plant model M with input search space S.
A region r.
The set P computed by the algorithm in Figure 6(a).
An objective function O.
An observation time T.
Output: The worst-case test scenario $testCase$.

1. $P' = \{(p, o) \in P \mid p \in r\}$
2. Let $(p, o) \in P'$ s.t. for all $(p', o') \in P'$, we have $o \geq o'$
3. $worstFound = o$
4. $AdaptParameteres(r, P)$
5. **for** K iterations **:**
6. $newp = Tweak(p)$
7. Let `Desired` be a step function generated by $newp$
8. Run M with the `Desired` input to obtain the `Actual` output
9. $v = O(\texttt{Desired}, \texttt{Actual})$
10. **if** $v > worstFound$ **:**
11. $worstFound = v$
12. $testCase = newp$
13. $p = Replace(p, newp)$
14. **return** $testCase$

(b)

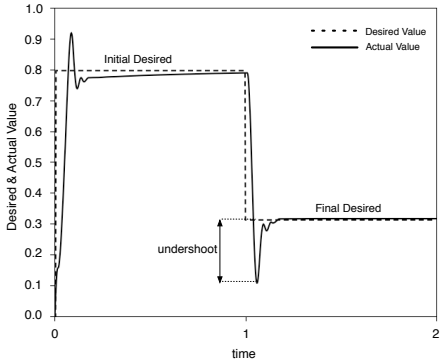

Figure 7: The second step of our approach in Figure 5: (a) The single-state search algorithm. (b) An example output diagram produced by the algorithm in (a)

time. Each fitness computation requires us to generate a T second simulation of the controller-plant model. This can take up to several minutes. For this reason, for the second step of our approach in Figure 5, we opt for a single-state search method in contrast to a population-based search such as Genetic Algorithms (GA) [13]. Note that in population-based approaches, we have to compute fitness functions for a set of points (a population) at each iteration.

The algorithm in Figure 7(a) represents a generic single-state search algorithm. Specifically, there are two placeholders in this figure: $Tweak()$ at line 6, and $Replace()$ at line 13. To implement different single-state search heuristics, one needs to define how the existing point p should be modified (the $Tweak()$ operator), and to determine when the existing point p should be replaced with a newly generated point $newp$ (the $Replace$ operator).

In our work, we instantiate the algorithm in Figure 7(a) based on three single-state search heuristics: standard Hill-Climbing (HC), Hill-Climbing with Random Restarts (HCRR), and Simulated Annealing (SA). Specifically, the $Tweak()$ operator for HC shifts p in the space by adding values x' and y' to the dimensions of p. The x' and y' values are selected from a normal distribution with mean $\mu = 0$ and variance σ^2. The $Replace$ operator for HC replaces p with $newp$, if and only if $newp$ has a worse (higher) objective function than p. HCRR and SA are different from HC only in their replacement policy. HCRR restarts the search from time to time by replacing p with a randomly selected point. Like HC, SA always replaces p with $newp$ if $newp$ has a worse (higher) objective function. However, SA may replace p with $newp$ even if $newp$ has a better (lower) objective function. This latter situation occurs only if another condition based on a random variable (temperature t) holds. Temperature is initialized to some value at the beginning of the search and decreases over time, meaning that SA replaces p with $newp$ more often at the beginning and less often towards the end of the

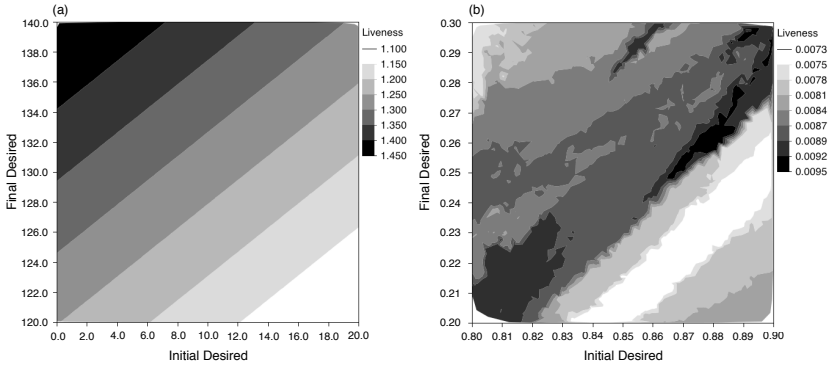

Figure 8: Diagrams representing the landscape for regular and irregular HeatMap regions: (a) A regular region with a clear gradient between the initial point of the search and the worst-case point. (b) An irregular region with several local optima.

search. That is, SA is more explorative during the first iterations, but becomes more and more exploitative over time.

In general, single-state search algorithms, including HC, HCRR, and SA, have a number of configuration parameters (e.g., variance σ in HC, and the initial temperature value and the speed of decreasing the temperature in SA). These parameters serve as knobs with which we can tune the degree of exploitation (or exploration) of the algorithms. To be able to effectively tune these parameters in our work, we visualized the landscape of several regions from our HeatMap diagrams. We noticed that the region landscapes can be categorized into two groups: (1) Regions with a clear gradient between the initial point of the search and the worst-case point (see e.g., Figure 8(a)). (2) Regions with a noisier landscape and several local optima (see e.g., Figure 8(b)). We refer to the regions in the former group as *regular regions*, and to the regions in the latter group as *irregular regions*. As expected, for regular regions, like the region in Figure 8(a), exploitative heuristics work best, while for irregular regions, like the region in Figure 8(b), explorative heuristics are most suitable [13].

Note that the number of points generated and evaluated in each region in the first step (the exploration step) is not sufficiently large so that we can conclusively determine whether a given region belongs to the regular group or to the irregular group above. Therefore, in our work, we rely on a heuristic that attempts to predict the region group based on the information available in HeatMap diagrams. Specifically, our observation shows that dark regions mostly surrounded by dark shaded regions belong to regular regions, while dark regions located in generally light shaded areas belong to irregular regions. Using this heuristic, we determine whether a given region belongs to a regular group or to an irregular group. For regular regions, we need to use algorithms that exhibit a more exploitative search behavior, and for irregular regions, we require algorithms that are more explorative. In Section 6, we evaluate our single-state search algorithms, HC, HCRR and SA, by applying them to both groups of regions, and comparing their performance in identifying worst-case scenarios in each region group.

5. Tool Support

We have fully automated and implemented our approach in a tool called CoCoTest (`https://sites.google.com/site/cocotesttool/`). CoCoTest implements the entire MiL testing process shown in Figure 5. Specifically, CoCoTest provides users with the following main functions: (1) Creating a workspace for testing a desired Simulink model. (2) Specifying the information about the input and output of the model under test. (3) Specifying the number of regions in a HeatMap diagram and the number of test cases to be run in each region. (4) Allowing engineers to identify the critical regions in a HeatMap diagram. (5) Generating HeatMap diagrams for each requirement. (6) Reporting a list of worst-case test scenarios for a number of regions. (7) Enabling users to run the model under test for any desired point in the input search space. In addition, CoCoTest can be run in a maintenance mode, allowing an advanced user to configure sophisticated features of the tool. This, in particular, includes choosing and configuring the algorithms used for the exploration and single-state search steps. Specifically, the user can choose between random search or adaptive random search for exploration, and between Hill-Climbing, Hill-Climbing with Random Restarts and Simulated Annealing for single-state search. Finally, the user can configure the specific parameters of each of these algorithms as discussed in Section 4.2.

As shown in Figure 5, the input to CoCoTest is a controller-plant model implemented in Matlab/Simulink and provided by the user. We have implemented the five generic objective functions discussed in Section 3.2 in CoCoTest. The user can retrieve the HeatMap diagrams and the worst-case scenarios for each of these objective functions separately. In addition, the user can specify the critical operating regions of a controller under test either by way of excluding HeatMap regions that are not of interest, or by including those that he wants to focus on. The worst-case scenarios can be computed only for those regions that the user has included, or has not excluded. The user also specifies the number of regions for which a worst-case scenario should be generated. CoCoTest sorts the regions that are picked by the user based on the results from the exploration step, and computes the worst-case scenarios for the ones that are top in the sorting depending on the number of worst-case scenarios requested by the user. The final output of CoCoTest is five HeatMap diagrams for the five objective functions specified in the tool, and a list of worst-case scenarios for each HeatMap diagram. The user can examine the HeatMap diagram and run worst-case test scenarios in Matlab individually. Further, the user can browse the HeatMap diagrams, pick any point in the diagram, and run the test scenario corresponding to that point in Matlab.

CoCoTest is implemented in Microsoft Visual Studio 2010 and Microsoft .NET 4.0. It is an object-oriented program in C# with 65 classes and roughly 30K lines of code. The main functionalities of CoCoTest have been tested with a test suite containing 200 test cases. CoCoTest requires Matlab/Simulink to be installed and operational on the same machine to be able to execute controller-plant model simulations. We have tested CoCoTest on Windows XP and Windows 7, and with Matlab 2007b and Matlab 2012b. Matlab 2007 was selected because Delphi Simulink models were compatible with this version of Matlab. We have made CoCoTest available to Delphi, and have presented it in a hands-on tutorial to Delphi function engineers.

6. Evaluation

In this section, we present the research questions that we set out to answer (Sections 6.1), relevant information about the industrial case study (Section 6.2), and the key parameters in setting our experiment and tuning our search algorithms (Section 6.3). We then provide answers to our research questions based on the results obtained from our experiment (Section 6.4). Finally, we discuss practical usability of the HeatMap diagrams and the worst-case test scenarios generated by our approach (Section 6.5).

6.1. Research Questions

RQ1: How does adaptive random search perform compared to naive random search in generating HeatMap diagrams?

RQ2: How do our single-state search algorithms (i.e., HC, HCRR, and SA) compare with one another in identifying the worst-case test scenarios? How do these algorithms compare with random search (baseline)?

RQ3: Does our single-state search algorithm (step 2 in Figure 5) improve the results obtained by the exploration step (step 1 in Figure 5)?

RQ4: Does our MiL testing approach help identify test cases that are useful in practice?

Any search-based solution should be compared with random search which is a standard "baseline" of comparison. If a proposed search-based solution does not show any improvement over random search, either something is wrong with the solution or the problem is trivial for which a random search approach is sufficient. In **RQ1** and **RQ2**, we respectively compare, with random search, our adaptive random search technique for HeatMap diagram generation, and our single-state search algorithms in finding worst-case test scenarios. In **RQ2**, in addition to comparing with random search, we compare our three single-state search algorithms with one another to identify if there is an algorithm that uniformly performs better than others for all the HeatMap regions. In **RQ3**, we argue that the second step of our approach (the search step) is indeed necessary and improves the results obtained during the exploration step considerably. In **RQ4**, we compare our best results, i.e., test cases with highest (worst) objective function values, with the existing test cases used in practice.

6.2. Case Studies

To perform our experiments, we applied our approach in Figure 5 to two case studies: A simple publicly available case study (DC Motor controller), and a real case study from Delphi (SBPC). Having one publicly available case study allows other researchers to compare their work with ours and to replicate our study. This is the main reason we included the DC Motor case study, even if it is simpler and less interesting than SBPC.

- **DC Motor Controller**: This case study consists of a Simulink PID Controller block (controller model) connected to a simple model of a DC Motor (plant model). The case study is taken from a Matlab/Simulink tutorial provided by MathWorks [16]. The controller model in this case study essentially controls the speed of a DC Motor. Specifically, the controller controls the voltage of the

Table 1: Size and complexity of our case study models.

Model features	**DC Motor** (M)		**SBPC** (M)	
	Controller	Plant	Controller	Plant
Blocks	8	13	242	201
Levels	1	1	6	6
Subsystems	0	0	34	21
Input Var.	1	1	21	6
Output Var.	1	2	42	7
LOC	150	220	8900	6700

DC Motor so that it reaches a desired angular velocity. Hence, the desired and actual values (see Figure 1(a)) represent the desired and actual angular velocities of the motor, respectively. The angular velocity of the DC Motor is a float value bounded within $[0...160]$.

- **Suppercharger Bypass Position Controller**: Supercharger is an air compressor blowing into a turbo-compressor to increase the air pressure supplied to the engine and, consequently, increase the engine torque at low engine speeds. The air pressure can be rapidly adjusted by a mechanical bypass flap. When the flap is completely open, supercharger is bypassed and the air pressure is minimum. When the flap is completely closed, the air pressure is maximum. **S***upercharger* **B***ypass Flap* **P***osition* **C***ontroller (SBPC)* is a component that determines the position of the bypass flap to reach to a desired air pressure. In SBPC, the desired and actual values (see Figure 1(a)) represent the desired and actual positions of the flap, respectively. The flap position is a float value bounded within $[0...1]$ (open when 0 and closed when 1.0).

The DC Motor controller, the SBPC controller, and their corresponding plant models are all implemented in Matlab/Simulink. Table 1 provides some metrics representing the size and complexity of the Simulink models for these two case studies. The table shows the number of Simulink blocks, hierarchy levels, subsystems, and input/output variables in each controller and in each plant model. In addition, we generated C code from each model using Matlab auto-coding tool [17], and have reported the (estimated) number of lines of code (excluding comments) generated from each model in the last row of Table 1.

Note that the desired and actual angular velocities of the DC Motor, and the desired and actual bypass flap positions are among the input and output variables of the controller and plant models. Recall that desired values are input variables to controller models, and actual values are output variables of plant models. SBPC models have several more input/output variables representing configuration parameters.

6.3. Experiments Setup.

Before running the experiments, we need to set the controller requirements parameters introduced in Figure 3. Table 2 shows the requirements parameter values, the observation time T used in our experiments, and the actual simulation times of our case study models. For SBPC, the requirements parameters were provided as part of the case study, but for DC Motor, we chose these parameters based on the maximum

Table 2: Requirements parameters and simulation time for the DC Motor and SBPC case studies

Requirements Parameters	**DC Motor**	**SBPC**
Liveness	$t_1 = 3.6s$	$t_1 = 0.8s$
Stability	$t_2 = 3.6s$	$t_2 = 0.8s$
Smoothness Normalized Smoothness	$v_1 = 8$	$v_1 = 0.05$
Responsiveness	$v_3 = 4.8$	$v_3 = 0.03$
Observation Time	$T = 8s$	$T = 2s$
Actual Simulation Running Time on Amazon	$50ms$	$31s$

Table 3: Parameters for the Exploration step

Parameters for Exploration	**DC Motor**	**SBPC**
Size of search space	$[0..160] \times [0..160]$	$[0..1] \times [0..1]$
HeatMap dimensions	8×8	10×10
Number of points per region (N in Figure 6(a))	10	10

value of the DC Motor speed. Specifically, T is chosen to be large enough so that the actual value can stabilize at the desired value. Note that as we discussed in Section 3.2, since we do not have pass/fail conditions, we do not specify v_2 from the smoothness and t_3 from the responsiveness properties.

We ran the experiments on Amazon micro instance machines which are equal to two Amazon EC2 compute units. Each EC2 compute unit has a CPU capacity of a 1.0-1.2 GHz 2007 Xeon processor. A single 8-second simulation of the DC Motor model and a single 2-second simulation of the SBPC Simulink model (e.g., Figure 7(b)) respectively take about 50 msec and 31 sec on the Amazon machine (See Table 2 last row).

We now discuss the parameters of the exploration and search algorithms in Figures 6(a) and 7(a). Table 3 summarizes the parameters we used in our experiment to run the exploration algorithm. Specifically, these include the size of the search space, the dimensions of the HeatMap diagrams, and the minimum number of points that are selected and simulated in each HeatMap region during exploration. Note that the input search spaces of both case studies are the set of floating point values within the search spaces specified in Table 3 first row.

We chose the HeatMap dimensions and the number of points per region, i.e., the value of N, (lines 2 and 3 in Table 3) by balancing and satisfying the following criteria: (1) The region shades should not change across different runs of the exploration algorithms. (2) The HeatMap regions should not be so fine grained such that we have to generate too many points during exploration. (3) The HeatMap regions should not be too coarse grained such that the points generated within one region have drastically different objective function values.

For both case studies, we decided to generate at least 10 points in each region during exploration ($N = 10$). We divided the search space into 100 regions in SBPC (10×10), and into 64 regions in DC Motor (8×8), generating a total of at least 1000 points

Table 4: Parameters for the Search step

Single-State Search Parameters		**DC Motor**	**SBPC**
Number of Iterations (HC, HCRR, SA)		100	100
Exploitative Tweak (σ) (HC, HCRR, SA)		2	0.01
Explorative Tweak (σ) (HC)		-	0.03
Distribution of Restart Iteration Intervals (HCRR)		$U(20, 40)$	$U(20, 40)$
Initial Temperature (SA)	Liveness	0.3660831	0.0028187
	Stability	0.0220653	0.000161
	Smoothness	12.443589	0.0462921
	Normalized Smoothness	0.08422266	0.1197671
	Responsiveness	0.0520161	0.0173561
Schedule (SA)	Liveness	0.0036245851	0.000027907921
	Stability	0.0002184	0.0000015940594
	Smoothness	0.12320385	0.00045833762
	Normalized Smoothness	0.00083388772	0.0011858129
	Responsiveness	0.00051501089	0.00017184257

and 640 points for SBPC and DC Motor, respectively. We executed our exploration algorithms a few times for SBPC and DC Motor case studies, and for each of our five objective functions. For each function, the region shades remained completely unchanged across the different runs. In all the resulting HeatMap diagrams, the points in the same region have close objective function values. On average, the variance over the objective function values for an individual region was small. Hence, we conclude that our selected parameter values are suitable for our case studies, and satisfy the above three criteria.

Table 4 shows the list of parameters for the search algorithms that we used in the second step of our work, i.e., HC, HCRR, and SA. Here, we discuss these parameters and justify the selected values in Table 4:

Number of Iterations (K): We ran each single-state search algorithm for 100 iterations, i.e., $K = 100$ in Figure 7(a). This is because the search has always reached a plateau after 100 iterations in our experiments. On average, iterating each of HC, HCRR, and SA for 100 times takes a few seconds for DC Motor and around one hour for SBPC on the Amazon machine. Note that the time for each iteration is dominated by the model simulation time. Therefore, the time required for our experiment was roughly equal to multiplying 100 by the time required for one single simulation of each model identified in Table 2 (50 msec for DC Motor and 31 sec for SBPC).

Exploitative and Explorative Tweak (σ): Recall that in Section 4.2, we discussed the need for having two Tweak operators: One for exploration, and one for exploitation. Specifically, each Tweak operator is characterized by a normal distribution with μ (mean) and σ (variance) values from which random values are selected. We set $\mu = 0$ in our experiment. For an exploitative Tweak, we choose $\sigma = 2.0$ for DC Motor, and $\sigma = 0.01$ for SBPC. As intended, with a probability of 99% the result of tweaking a point in the center of a HeatMap region stays inside that region in both case studies. Obviously, this probability decreases when

the point moves closer to the borders. In our search, we discard the result of Tweak when it generates points outside of the regions, and never generate simulations for them. In addition, with these values for σ, the search tends to be exploitative. Specifically, the Tweak has a probability of 70% to modify individual points' dimensions within a range defined by σ.

To obtain an explorative Tweak operator, we triple the above values for σ. Note that in our work, we use the explorative Tweak option only with the HC algorithm. HCRR and SA are turned into explorative search algorithms using *restart* and *temperature* options discussed below. In addition, in the DC Motor case study, we do not need an explorative Tweak operator because all the HeatMap regions belong to regular regions for which an exploitative Tweak is expected to work best.

Restart for HCRR: HCRR is similar to HC 3except that from time to time it restarts the search from a new point in the search space. For this algorithm, we need to determine how often the search is restarted. In our work, the number of iterations between each two consecutive restarts is randomly selected from a uniform distribution between 20 and 40, denoted by $U(20, 40)$.

Initial Temperature and Schedule: The SA algorithm requires a temperature that is initialized at the beginning of the search, and is incremented iteratively based on the value of a schedule parameter. The values for the temperature and schedule parameters should satisfy the following criteria [13]: (1) The initial value of temperature should be comparable with differences between the objective function values of pairs of points in the search space. (2) The temperature should converge towards zero without reaching it. We set the initial value of temperature to be the standard deviation of the objective function values computed during the exploration step. The schedule is then computed by dividing the initial value of temperature by 101, ensuring that the final value of temperature after 100 iterations does not become equal to zero.

6.4. Results Analysis

RQ1. How does adaptive random search perform compared to naive random search in generating HeatMap diagrams? To answer this question, we compare (1) the HeatMap diagrams generated by naive random search and adaptive random search, and (2) the time these two algorithms take to generate their output HeatMap diagrams.

For each of our case studies, we compared three HeatMap diagrams randomly generated by naive random search with three HeatMap diagrams randomly generated by adaptive random search. Specifically, we compared the region colors and value ranges related to each color. We noticed that all the HeatMap diagrams related to DC Motor (resp. SBPC) were similar. Hence, we did not observe any differences between these two algorithms by comparing their generated HeatMap diagrams.

Figures 9 and 10 represent example sets of HeatMap diagrams generated for DC Motor and SBPC case studies, respectively. In each figure, there are five diagrams corresponding to our five objective functions. Note that as we discussed above, because

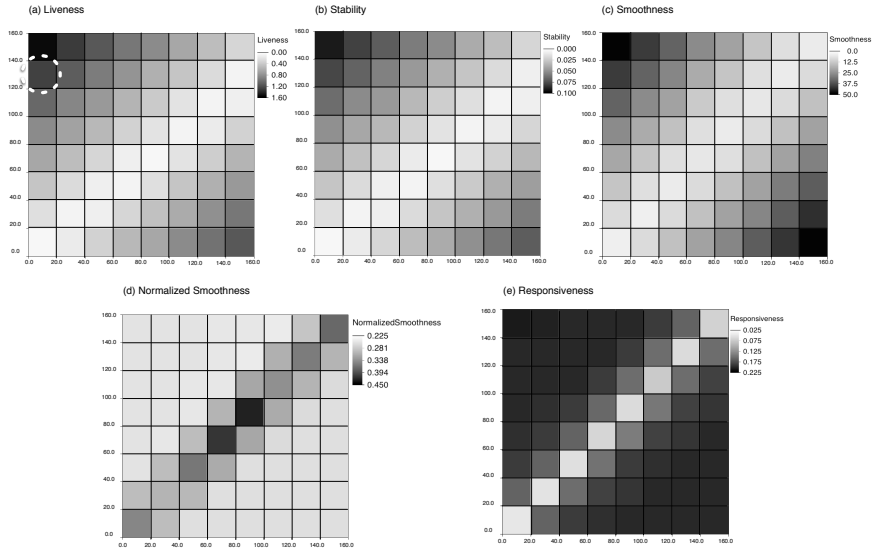

Figure 9: HeatMap diagrams generated for DC Motor for the (a) Liveness, (b) Stability, (c) Smoothness, (d) Normalized Smoothness and (e) Responsiveness requirements. Among the regions in these diagrams, we applied our single-state search algorithms to the region specified by a white dashed circle.

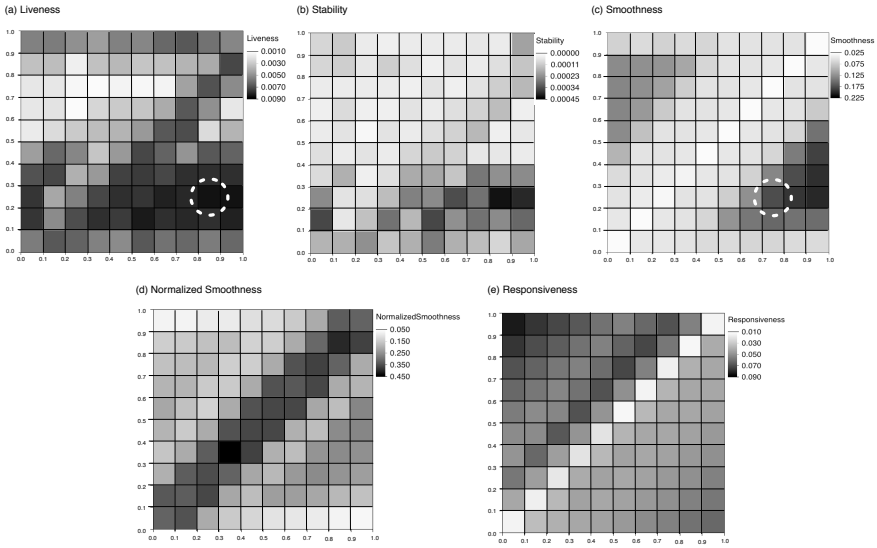

Figure 10: HeatMap diagrams generated for SBPC for the (a) Liveness, (b) Stability, (c) Smoothness, (d) Normalized Smoothness and (e) Responsiveness requirements. Among the regions in these diagrams, we applied our single-state search algorithms to the two regions specified by white dashed circles.

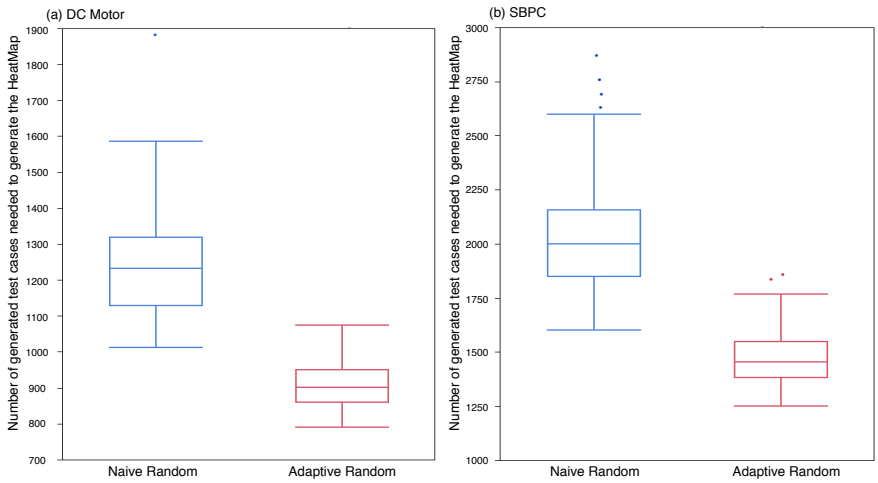

Figure 11: Comparing execution times of naive random search and adaptive random search algorithms for HeatMap diagram generation (the Exploration step).

the HeatMap diagrams generated by random search and adaptive random search were similar, we have shown only one set of diagrams for each case study here.

In order to compare the speed of naive random and adaptive random search algorithms in generating HeatMap diagrams, we ran both algorithms 100 times to account for their randomness. For each run, we recorded the number of iterations that each algorithm needed to generate at least N points in each HeatMap region. Figure 11(a) shows the distributions of the number of iterations obtained by applying these two algorithms to DC Motor, and Figure 11(b) shows the distributions obtained when the algorithms are applied to SBPC. As shown in the figures, in both case studies, adaptive random search was considerably faster (required fewer iterations) than naive random search. On average, for DC Motor, it took 1244 iterations for naive random search to generate HeatMap diagrams, while Adaptive random search required 908 iterations on the same case study. For SBPC, the average number of iterations for naive random search and adaptive random search were 2031 and 1477, respectively.

Recall from Table 2 that a single model simulation for DC Motor takes about 50 ms. Hence, an average run of adaptive random search is about a few seconds faster than that of naive random search for DC Motor. This speed difference significantly increases for SBPC, which is a more realistic case, where the average running time of naive random search is about five hours more than that of adaptive random search. Based on results, given realistic cases and under time constraints, adaptive random search allows us to generate significantly more precise HeatMap diagrams (if needed), within a shorter time.

RQ2. How do our single-state search algorithms (i.e., HC, HCRR, and SA) compare with one another in identifying the worst-case test scenarios? How do these algorithms compare with random search (baseline)? To answer this question, we compare the performance of our three different single-state search algorithms, namely HC, HCRR, and SA, which we used in the search step of our framework. In addi-

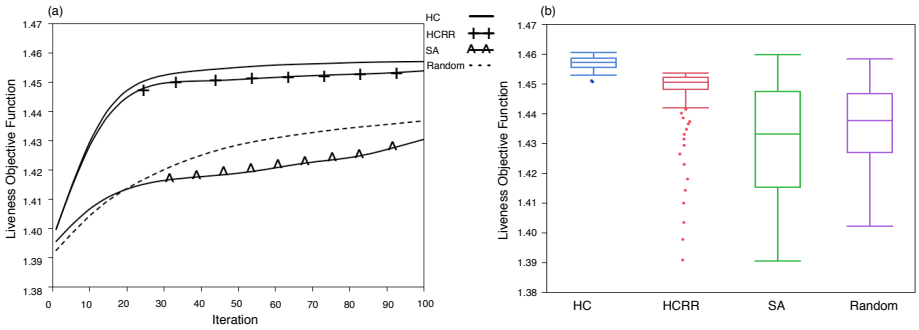

Figure 12: The result of applying HC, HCRR, SA and random search to a regular region from DC Motor (specified by a dashed white circle in Figure 9(a)): (a) The averages of the output values (i.e., the highest objective function values) of each algorithm obtained across 100 different runs over 100 iterations. (b) The distributions of the output values obtained across 100 different runs of each algorithm after completion, i.e., at iteration 100.

tion, we compare these algorithms with random search which is used as a baseline of comparison for search-based algorithms. Recall that in section 4.2, we identified two different groups of HeatMap regions. Here, we compare the performance of HC, HCRR, SA, and random search in finding worst-case test scenarios for both groups of HeatMap regions, separately. As shown in Figure 7(a) and mentioned in section 4.2, for each region, we start the search from the worst point found in that region during the exploration step. This not only allows us to reuse the results from the exploration step, but also makes it easier to compare the performance of the single-state search algorithms as these algorithms all start the search from the same point in the region and the same objective function value.

We selected three regions from the set of high risk regions of each one of the HeatMap diagrams in Figures 9 and 10, and applied HC, HCRR, SA, and random search to each of these regions. In total, we applied each single-state search algorithm to 15 regions from DC Motor and to 15 regions from SBPC. For DC Motor, we selected the three worst (darkest) regions in each HeatMap diagram, and for SBPC case study, the domain expert chose the three worst (darkest) regions among the critical operating regions of the SBPC controller.

We noticed that all the HeatMap regions in the DC Motor case study were from group regular with a clear gradient from light to dark. Therefore, for the DC Motor regions, we ran our single-state search algorithms only with the exploitative parameters in Table 4. For the SBPC case study, we ran the search algorithms with the exploitative parameters for nine regions (regions from smoothness, normalized smoothness, and responsiveness HeatMap diagrams), and with the explorative parameters for six other regions (regions from liveness and stability HeatMap diagrams).

Figure 12 shows the results of comparing our three single-state search algorithms as well as random search using a representative region from the DC Motor case study. Specifically, the results in this figure were obtained by applying these algorithms to the region specified by a white dashed circle in Figure 9(a). The results of applying our algorithms to the other 14 HeatMap regions selected from the DC Motor case study were similar. As before, we ran each of these algorithms 100 times. Figure 12(a) shows the averages of the highest (worst) objective function values for 100 different runs of

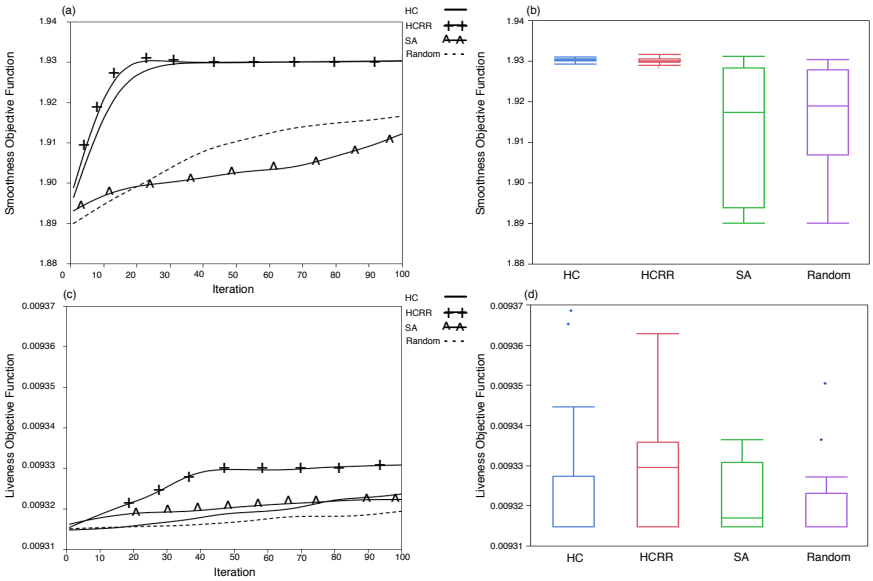

Figure 13: The result of applying HC, HCRR, SA and random search to a regular and an irregular region from SBPC: (a) and (c) show the averages of the output values (i.e., the highest objective function values) of each algorithm obtained across 100 different runs over 100 iterations. (b) and (d) show the distributions of the output values obtained across 100 different runs of each algorithm over 100 iterations. Diagrams (a,b) are related to the region specified by a dashed white circle in Figure 10(c), and Diagrams (c,d) are related to the region specified by a dashed white circle in Figure 10(a).

each of our algorithms over 100 iterations. Figure 12(b) compares the distributions of the highest (worst) objective function values produced by each of our algorithms on completion (i.e., at iteration 100) over 100 runs.

Figure 13 represents the results of applying our algorithms to two representative HeatMap regions from the SBPC case study. Specifically, Figures 13(a) and (b) show the results related to a regular region, (i.e., regions with clear gradients from light to dark), and Figures 13(c) and (d) represent the results related to an irregular region, (i.e., regions with several local optima and no clear gradient landscape). The former region is from Figure 10(c), and the latter is from Figure 10(a). Both regions are specified by a white dashed circle in Figures 10 (c) and (a), respectively. For the regular region, we executed HC, HCRR, and SA with the exploitative tweak parameter in Table 4, and for the irregular region, we used the explorative tweak parameter for HC from the same table.

Similar to Figure 12(a), Figures 13(a) and (c) represent the averages of the algorithms' output values obtained from 100 different runs over 100 iterations, and similar to Figure 12(b), Figures 13(b) and (d) represent the distributions of the algorithms' output values obtained from 100 different runs. As before, the results in Figures 13(a) and (b) were representative of the results we obtained from other eight SBPC regular regions in our experiment, while the results in Figures 13(c) and (d) were representative

for the other five SBPC irregular regions.

The diagrams in Figures 12 and 13 show that HC and HCRR perform better than random search and SA for regular regions. Specifically, these two algorithms require fewer iterations to find better output values, i.e., higher objective function values, compared to SA and random search. HC performs better than HCRR on the regular region from DC Motor. As for the irregular region from SBPC, HCRR performs better than other algorithms.

To statistically compare the algorithms, we performed a *two-tailed, student t-test* [18] and other non-parametric equivalent tests, but only report the former as results were consistent. The chosen level of significance (α) was set to 0.05. Applying t-test to the regular region from DC Motor (Figures 12 (a) and (b)) resulted in the following order for the algorithms, from best to worst, with significant differences between all pairs: HC, HCRR, random search, and SA. The statistical test for both regular and irregular regions from SBPC (Figures 13(a) to (d)) showed that the algorithms are divided into two equivalence classes: (A) HC and HCRR, and (B) random search and SA. The p-values between pairs of algorithms in the same class are above 0.05, and the p-values between those in different classes are below 0.05. Table 5 provides the p-values comparing HC with other algorithms for regular regions, and comparing HCRR with other algorithms for an irregular region from SBPC. In addition, to computing p-values, we provide effect sizes comparing HC and HCRR with other algorithms for regular and irregular regions, respectively. In [19], it was noted that it is inadequate to merely show statistical significance alone. Rather we need to determine whether the effect size is noticeable. Therefore, in Table 5, we show effect sizes computed based on Cohen's d [20]. The effect sizes are considered small for $0.2 \leq d < 0.5$, medium for $0.5 \leq d < 0.8$, and high for $d \geq 0.8$.

To summarize, for regular regions exploitative algorithms work best. Specifically, HC, which is the most exploitative algorithm, performs better than other algorithms on regular regions from DC Motor. For the regular region from SBPC, HCRR manages to be as good as HC because it can reach its plateau early enough before it is restarted from a different point. Hence, HCRR and HC perform the same on the regular regions from SBPC.

For irregular regions, search algorithms that are more explorative do better. Specifically, HCRR is slightly better than HC on irregular regions from SBPC. The histogram diagrams in Figure 14 compare the distributions of the highest objective function values found by HC and HCRR. Specifically, with 50% probability, HCRR finds an objective function value larger than 0.00933. If we run HCRR for three times, with a probability of 87.5%, HCRR finds at least one value around or higher than 0.00933. For HC, however, the probability of finding an objective function value higher than 0.00933 is less than 20% in one run, and less than 49% in three runs. That is, even though we do not observe a statistically significant difference between the results of HC and HCRR for the irregular region of SBPC, by running these algorithms three times, we have a higher chance to find a larger value by HCRR than by HC. Finally, even though we chose SA parameters according to the guidelines in the literature (see Section 6.3, and [13]), on our case study models, SA is never better than random. This can be due to the fact that SA merely explores the space at the beginning of its search and becomes totally exploitative at the end of its search, but the exploitation is not necessarily performed

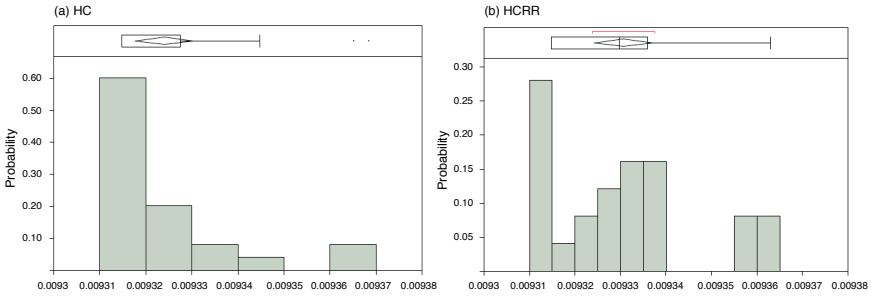

Figure 14: The distribution of the highest objective function values found after 100 iterations by HC and HCRR related to the region specified by a dashed white circle in Figure 10(a)

Table 5: The p-values obtained by applying t-test to the results in Figures 12 and 13 and the effect sizes measuring the differences between these results: Each algorithm is compared with HC (i.e., the best candidate for regular regions) for the regular DC Motor and SBPC regions, and with HCRR (i.e., the best candidate for irregular regions) for the irregular SBPC region.

Algorithm		**DC Motor Regular Region**	**SBPC Regular Region**	**SBPC Irregular Region**
p-value with HC (effect size)	HCRR	<0.0001(High)	0.2085(Low)	-
	SA	<0.0001(High)	<0.0001(High)	-
	Random	<0.0001(High)	<0.0001(High)	-
p-value with HCRR (effect size)	HC	-	-	0.0657(Medium)
	SA	-	-	0.0105(High)
	Random	-	-	0.0014(High)

close to a local optimum. However, HCRR periodically spends a number of iterations improving its candidate solution (exploitation) after each random restart (exploration). As a result, HCRR is more likely to perform some exploitation in parts of the search space close to a local optimum.

RQ3. Does our single-state search algorithm (step 2 in Figure 5) improve the results obtained by the exploration step (step 1 in Figure 5)? To answer this question, we compare the output of single-state search algorithm with the output of exploration step for two regular regions from DC Motor and SBPC, and one irregular region from SBPC. Relying on RQ1 and RQ2 results, for this comparison, we use the adaptive random search algorithm for the exploration step, HC for the single-state search in regular regions, and HCRR for the single-state search in irregular regions.

Let A be the highest objective function value computed by adaptive random search in each region, and let B_i be the output (highest objective function value) of HC and HCRR at run i on regular and irregular regions, respectively. We compute the relative improvement that the search step could bring about over the results of the exploration step for run i of the search by $\frac{B_i - A}{A}$. Figure 15 shows the distribution of these relative improvements for the three selected regions and across 100 different runs.

The results show that the final test cases computed by our best single-state search algorithm have higher objective function values compared to the best test cases identified by adaptive random search during the exploration step. The average relative improvement, for regular regions, is around 5.5% for DC Motor, and 7.8% for SBPC, which is a larger and more realistic case study and has a more complex search space.

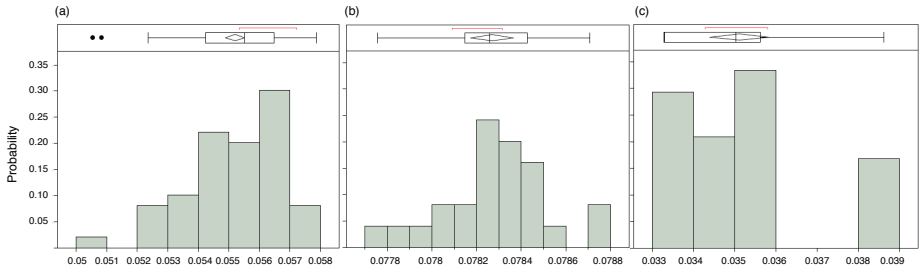

Figure 15: The distribution of the improvements of the single-state search output compared to the exploration output across 100 different runs of the search algorithm: (a) and (b) show the improvements obtained by applying HC to two regular regions from DC Motor and SBPC, and (c) shows the improvements obtained by applying HCRR to an irregular region from SBPC. The results are related to the three regions specified by dashed white circles in Figures 9(a), 10(a), and 10(c), respectively.

This value, for the irregular region form SBPC is about 3.5%.

RQ4. Does our MiL testing approach help identify test cases that are useful in practice? To demonstrate practical usefulness of our approach, we show that the test cases generated by our MiL testing approach had not been previously found by manual testing based on domain expertise. Specifically in our industry-strength case study (SBPC), we were able to generate 15 worst-case test scenarios for the SBPC controller. Figure 7(b) shows the simulation for one of these test scenarios concerning smoothness. Delphi engineers reviewed the simulation diagrams related to these worst-case scenarios to determine which ones are acceptable as they exhibit small deviations that can be tolerated in practice, and which ones are critical and have to be further investigated. Among the 15 worst-case test scenarios, those related to liveness and stability requirements were considered acceptable by the domain expert. The other nine test scenarios, however, indicated violations of the controller requirements. None of these critical test cases had been previously found by manual, expertise-based MiL testing by Delphi engineers. For example, Figure 7(b) shows an undershoot scenario around 0.2 for the SBPC controller. The maximum undershoot/overshoot for the SBPC controller identified by manual testing was around 0.05. Similarly, for the responsiveness property, we found a scenario in which it takes $150ms$ for the actual value to get close enough to the desired value while the maximum corresponding value in manual testing was around $50ms$.

6.5. Practical Usability

To better understand practical usability of our work, we made our case study results and tool support available to Delphi engineers through interactive tutorial sessions and our frequent meetings with them. In general, the engineers believe that our approach can help them effectively identify bugs in their controller models, and in addition, can be seen as a significant aid in designing controllers.

To receive feedback on specific output items of our work, we presented HeatMap diagrams shown in Figure 10 to Delphi engineers. They found the diagrams visually appealing and useful. They noted that the diagrams, in addition to enabling the identification of critical regions, can be used in the following ways: (1) The engineers can gain confidence about the controller behaviors over the light shaded regions of the diagrams. (2) The diagrams enable the engineers to investigate potential anomalies in

the controller behavior. Specifically, since controllers have continuous behaviors, we expect a smooth shade change over the search space going from white to black. A sharp contrast such as a dark region immediately neighboring a light shaded region may potentially indicate an abnormal behavior that needs to be further investigated.

As discussed in response to **RQ4**, we identified 15 worst case scenarios (test cases) for SBPC in total. Nine out of these 15 test cases indicated requirements violations at the MiL level. According to Delphi engineers, the violations revealed by our nine test cases could be due to a lack of precision or errors in the controller or plant models. In order to determine whether these MiL level errors arise in more realistic situations, Delphi engineers applied these nine test scenarios at the Hardware-in-the-Loop (HiL) level where the MiL level plant model is replaced with a combination of hardware devices and more realistic HiL plant models running on a real-time simulator. This study showed that:

- Three (out of nine) errors that were related to the responsiveness requirement disappeared at the HiL level. This indicates that the responsiveness MiL level errors were due to lack of precision or abstractions made in the plant model, as they do not arise in a more realistic HiL environment.
- Six (out of nine) errors that were related to the smoothness and normalized smoothness requirements repeated at the HiL level. Since Delphi engineers were not able to identify defects in the controller model causing these errors, they conjecture that these errors might be due to configuration parameters of the controllers, and disappear once the controller is configured with proper parameters taken from real cars. As discussed in Section 3.1, we do not focus on controller configuration parameters in this paper.

In addition, we also applied at the HiL level the six test cases out of the original 15 test cases that had passed the MiL testing stage. These six test cases were related to liveness and stability. The HiL testing results for these test cases were consistent with those obtained on MiL. That is, these test cases passed the HiL testing stage as well. To summarize, our approach is effective in finding worst-case scenarios that cannot be identified manually. Furthermore, such scenarios constitute effective test cases to detect potential errors in the controller model or in the plant model or in controller configuration parameters.

7. Related Work

Testing continuous controllers presents a number of challenges, and is not yet fully supported by existing tools and techniques [4, 3, 1, 21]. There are many approaches to testing and analysis of MATLAB/Simulink models [21]. We discuss these approaches and compare them with our work under the following categories:

Formal methods. The modeling languages that have been developed to capture embedded software systems mostly deal with discrete-event or mixed discrete-continuous systems [6, 22, 8]. Examples of these languages include timed automata [23], hybrid automata [7, 24], and Stateflow [25]. Automated reasoning tools built for these languages largely rely on formal methods. Specifically, these tools either exhaustively

verify the model under analysis, e.g., using model checkers [4, 26], or generate test cases from counter-examples produced by model checkers [27, 28, 10]. None of these tools, however, are directly applicable to Simulink models, and they require a pre-requisite translation step to convert simulink models into a formal intermediate notation (e.g., hybrid automata or the SAL notation [29]) [10]. Existing translation mechanisms apply to a subset of the Simulink notation and cannot handle the entire Simulink blocks [10]. In general, formal methods are more amenable to verification of logical and state-based behaviors such as invariance and reachability properties. Further, their scalability to large and realistic systems is still unknown. In our work, we focused on testing pure continuous controllers which have not previously been captured by any discrete-event or formal mixed discrete-continuous notation. Moreover, we demonstrated usefulness and scalability of our approach by evaluating it on a representative industrial case study.

Search-based approaches. Search-based techniques have been applied to Matlab/Simulink models for two different purposes:

1. Generation of complex input signals that are as close as possible to real-world signals, and can be used for testing continuous aspects of Simulink models [30]. Such signals can be generated either by sequencing parameterized base signals [31, 32], or by modifying parameters of Fourier series [32]. Meta-heuristic search algorithms are used to build such close-to-real-world signals by composing simpler input signals where the search objective is to match a user-defined description of the signal [31] or to satisfy certain signal constraints specified in temporal logic [33]. This approach does not address generation of test cases with respect to system requirements. Neither does it provide any insight as to how one can develop test oracles for the generated input signals. In our work, we focus on continuous controllers for which step functions are used as input signals. In addition, we derive test cases and test objectives from system requirements, enabling engineers to find worst-case test scenarios and develop test oracles based on each requirement.

   There has been also some approaches to automated signal analysis where simulation outputs of Simulink models are verified against customized *boolean* properties implemented via Matlab blocks [34]. Such boolean properties fail to capture complex requirements such as smoothness or responsiveness. Hence, in our work, we use *quantitative* objective functions over controller outputs. In addition, the signal analysis method in [34] does neither address systematic testing, nor does it include identification of test cases or formalization of test oracles.

2. Generation of test input data with the goal of maximizing coverage or adequacy criteria of test suites [35, 36]. Different meta-heuristic search techniques have been used to maximize coverage or adequacy criteria for MATLAB/Simulink models [36], e.g., they are used to maximize path coverage of Simulink models [37, 38], or the number of killed mutants by test cases [39], or coverage of the Stateflow statecharts [9]. This approach does not take into account system requirements. In addition, the coverage/adequacy criteria listed above do not apply

to continuous blocks of MATLAB/Simulink models. Finally, coverage/adequacy criteria satisfaction alone is a poor indication of test suite effectiveness [40]. Our work enables generation of test cases based on controller requirements. Moreover, our approach is not limited to MATLAB/Simulink and is applicable to testing continuous controllers, implemented in any modelling or programming language.

Control Theory. Continuous controllers have been widely studied in the control theory domain [2, 5, 41], where the focus has been to optimize the controller behavior for a specific application by design optimization [42], or a specific hardware configuration by configuration optimization [5]. Overall existing work in control theory deals with optimizing the controller design or configuration rather than testing. They normally check and optimize the controller behavior over one, or a few number of test cases and cannot substitute systematic testing as addressed by our approach.

Commercial Tools. Finally, a number of commercial verification and testing tools have been developed, aiming to generate test cases for MATLAB/Simulink models. The Simulink Design Verifier software [11] and Reactis Tester [12] are perhaps the most well-known tools but there are a few other competitors as well [43]. To evaluate requirements using these tools, the MATLAB/Simulink models need to be augmented with boolean assertions. The existing assertion checking mechanism, however, handles combinatorial and logical blocks only, and fails to evaluate the continuous MATLAB/Simulink blocks (e.g., integrator blocks) [22]. As for the continuous behaviors, these tools follow a methodology that considers the MiL models to be the test oracles [22]. Under this assumption, MiL level testing can never identify the discrepancies between controller-plant models and their high-level requirements. In our work, however, we rely on controller requirements as test oracles, and are able to identify requirement violations in the MiL level models. Another major issue about these commercial tools is a lack of detailed publicly available and research-based documentation [44, 45].

8. Conclusions

Testing and verification of the software embedded into cars is a major challenge in automotive industry. Software production in the automotive domain typically comprises three stages: Developing automotive control functions as Simulink models, generating C code from the Simulink models, and deploying the resulting code on hardware devices. The artifacts produced at each stage should be sufficiently tested before moving to the subsequent stage. Hence, automotive software artifacts are subject to (at least) the following three rounds of testing: Model-in-the-Loop (MiL) testing, Software-in-the-Loop (SiL) testing, and Hardware-in-the-Loop (HiL) testing. In this article, we proposed a search-based approach to automate generation of Model-in-the-loop (MiL) level test cases for continuous controllers. These controllers make up a large part of automotive functions. We identified and formalized a set of common requirements for this class of controllers. Our proposed technique relies on a combination of explorative and exploitative search algorithms, which aim at finding worst-case

scenarios in the input space with respect to the controller requirements. Our technique is implemented in a tool, named CoCoTest. We evaluated our approach by applying it to an automotive air compressor module and to a publicly available controller model. Our experiments showed that our approach automatically generates several worst-case scenarios, which can be used for testing purposes, that had not been previously found by manual testing based on domain expertise. The test cases indicated potential violations of the requirements at the MiL level, and were applied by Delphi engineers at the HiL level to identify potential discrepancies between plant models, and the HiL plant model and hardware. In addition, we demonstrated the effectiveness and efficiency of our search strategy by showing that our approach computes significantly better test cases and is significantly faster than a pure random test case generation strategy.

In future, we plan to perform more case studies with various controllers and from different domains to demonstrate generalizability and scalability of our work. In addition, we want to extend our approach to include configuration parameters of continuous controllers in our test case generation strategy. To do so, we need to find effective and scalable techniques that can search through large and multi-dimensional search spaces, and effectively help engineers visualize such spaces. Furthermore, we intend to evaluate our tool, CoCoTest, to assess its abilities and performance in bug finding and fault localization of Simulink models. Extending our search-based testing approach to other types of embedded Simulink components such as state machine controllers and components calculating physical properties is another avenue for future work. Finally, in collaboration with Delphi engineers, we intend to empirically and systematically evaluate our tool with respect to detecting and localizing real-world faults in Simulink models.

Acknowledgments

Supported by the Fonds National de la Recherche - Luxembourg (FNR/P10/03 and FNR 4878364), and Delphi Automotive Systems, Luxembourg.

References

[1] J. Zander, I. Schieferdecker, P. J. Mosterman, Model-based testing for embedded systems, volume 13, CRC Press, 2012.

[2] N. S. Nise, Control Systems Engineering, 4th ed., John-Wiely Sons, 2004.

[3] E. Lee, S. Seshia, Introduction to Embedded Systems: A Cyber-Physical Systems Approach, http://leeseshia.org, 2010.

[4] T. Henzinger, J. Sifakis, The embedded systems design challenge, in: FM, 2006, pp. 1–15.

[5] M. Araki, PID control, Control systems, robotics and automation 2 (2002) 1–23.

[6] A. Pretschner, M. Broy, I. Krüger, T. Stauner, Software engineering for automotive systems: A roadmap, in: FOSE, 2007, pp. 55–71.

[7] T. Henzinger, The theory of hybrid automata, in: LICS, 1996, pp. 278–292.

[8] T. Stauner, Properties of hybrid systems-a computer science perspective, Formal Methods in System Design 24 (2004) 223–259.

[9] A. Windisch, Search-based test data generation from stateflow statecharts, in: Proceedings of the 12th annual conference on Genetic and evolutionary computation, ACM, 2010, pp. 1349–1356.

[10] S. Mohalik, A. A. Gadkari, A. Yeolekar, K. Shashidhar, S. Ramesh, Automatic test case generation from simulink/stateflow models using model checking, Software Testing, Verification and Reliability (2013). to appear.

[11] The MathWorks Inc., Simulink, http://www.mathworks.nl/products/simulink, 2003. [Online; accessed 25-Nov-2013].

[12] Reactive Systems Inc., http://www.reactive-systems.com/simulink-testing-validation.html, 2010. [Online; accessed 25-Nov-2013].

[13] S. Luke, Essentials of Metaheuristics, Lulu, 2009. http://cs.gmu.edu/~sean/book/metaheuristics/.

[14] G. Grinstein, M. Trutschl, U. Cvek, High-dimensional visualizations, in: 7th Workshop on Data Mining Conference KDD Workshop, 2001, pp. 7–19.

[15] The MathWorks Inc., Matlab quasi random numbers, http://www.mathworks.nl/help/stats/generating-quasi-random-numbers.html, 2003. [Online; accessed 17-Mar-2014].

[16] The MathWorks Inc., DC Motor Simulink Model, http://www.mathworks.com/matlabcentral/fileexchange/11587-dc-motor-model-simulink, 2009. [Online; accessed 25-Nov-2013].

[17] The MathWorks Inc., Embedded Coder, http://www.mathworks.nl/products/embedded-coder/, 2011. [Online; accessed 25-Nov-2013].

[18] J. Capon, Elementary Statistics for the Social Sciences: Study Guide, Wadsworth Publishing Company, 1991.

[19] A. Arcuri, L. Briand, A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering, Software Testing, Verification and Reliability (2012).

[20] J. Cohen, Statistical power analysis for the behavioral sciences (rev, Lawrence Erlbaum Associates, Inc, 1977.

[21] F. Elberzhager, A. Rosbach, T. Bauer, Analysis and testing of matlab simulink models: A systematic mapping study, in: Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to testing Automation (JAMAICA'13), ACM, 2013, pp. 29–34.

[22] P. Skruch, M. Panek, B. Kowalczyk, Model-based testing in embedded automotive systems, Model-Based Testing for Embedded Systems (2011) 293–308.

[23] R. Alur, Timed automata, in: CAV, 1999, pp. 8–22.

[24] J.-F. Raskin, An introduction to hybrid automata, in: Handbook of networked and embedded control systems, Springer, 2005, pp. 491–517.

[25] A. Sahbani, J. Pascal, Simulation of hyibrd systems using stateflow, in: ESM, 2000, pp. 271–275.

[26] M. Mazzolini, A. Brusaferri, E. Carpanzano, Model-checking based verification approach for advanced industrial automation solutions, in: IEEE Conference on Emerging Technologies and Factory Automation (ETFA), 2010, IEEE, 2010, pp. 1–8.

[27] A. A. Gadkari, A. Yeolekar, J. Suresh, S. Ramesh, S. Mohalik, K. Shashidhar, Automotgen: Automatic model oriented test generator for embedded control systems, in: Computer Aided Verification, Springer, 2008, pp. 204–208.

[28] M. Satpathy, A. Yeolekar, P. Peranandam, S. Ramesh, Efficient coverage of parallel and hierarchical stateflow models for test case generation, Software Testing, Verification and Reliability 22 (2012) 457–479.

[29] Symbolic Analysis Laboratory Homepage, http://sal.csl.sri.com, 2001. [Online; accessed 25-Nov-2013].

[30] T. E. Vos, F. F. Lindlar, B. Wilmes, A. Windisch, A. I. Baars, P. M. Kruse, H. Gross, J. Wegener, Evolutionary functional black-box testing in an industrial setting, Software Quality Journal 1 (2013) 1–30.

[31] A. Baresel, H. Pohlheim, S. Sadeghipour, Structural and functional sequence test of dynamic and state-based software with evolutionary algorithms, in: Genetic and Evolutionary Computation—GECCO 2003, Springer, 2003, pp. 2428–2441.

[32] A. Windisch, N. Al Moubayed, Signal generation for search-based testing of continuous systems, in: International Conference on Software Testing, Verification and Validation Workshops, 2009. ICSTW'09., IEEE, 2009, pp. 121–130.

[33] B. Wilmes, A. Windisch, Considering signal constraints in search-based testing of continuous systems, in: Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW), 2010, IEEE, 2010, pp. 202–211.

[34] J. Zander-Nowicka, Model-based Testing of Real-Time Embedded Systems in the Automotive Domain, Ph.D. thesis, Elektrotechnik und Informatik der Technischen Universitat, Berlin, 2009.

[35] Y. Zhan, J. A. Clark, A search-based framework for automatic testing of matlab/simulink models, Journal of Systems and Software 81 (2008) 262–285.

[36] K. Ghani, J. A. Clark, Y. Zhan, Comparing algorithms for search-based test data generation of matlab simulink models, in: IEEE Congress on Evolutionary Computation, 2009. CEC'09., IEEE, 2009, pp. 2940–2947.

[37] Y. Zhan, J. Clark, Search based automatic test-data generation at an architectural level, in: Genetic and Evolutionary Computation–GECCO 2004, Springer, 2004, pp. 1413–1424.

[38] Y. Zhan, J. A. Clark, The state problem for test generation in simulink, in: Proceedings of the 8th annual conference on Genetic and evolutionary computation, ACM, 2006, pp. 1941–1948.

[39] Y. Zhan, J. A. Clark, Search-based mutation testing for simulink models, in: Proceedings of the 2005 conference on Genetic and evolutionary computation, ACM, 2005, pp. 1061–1068.

[40] M. Staats, G. Gay, M. Whalen, M. Heimdahl, On the danger of coverage directed test case generation, in: Fundamental Approaches to Software Engineering, Springer, 2012, pp. 409–424.

[41] T. Wescott, PID without a PhD, Embedded Systems Programming 13 (2000) 1–7.

[42] Wikipedia, PID controller, http://en.wikipedia.org/wiki/PID_controller, 2004. [Online; accessed 25-Nov-2013].

[43] Berner and Mattner Inc., http://www.berner-mattner.com/de/berner-mattner-home/unternehmen/index.html, 2011. [Online; accessed 25-Nov-2013].

[44] S. Sims, D. C. DuVarney, Experience report: the reactis validation tool, ACM SIGPLAN Notices 42 (2007).

[45] J. Wegener, P. M. Kruse, Search-based testing with in-the-loop systems, in: 1st International Symposium on Search Based Software Engineering, 2009, IEEE, 2009, pp. 81–84.