CoCoTest: A Tool for Model-in-the-Loop Testing of Continuous Controllers

Reza Matinnejad, Shiva Nejati, Lionel C. Briand
SnT Centre, University of Luxembourg, Luxembourg
{reza.matinnejad,shiva.nejati,lionel.briand}@uni.lu

Thomas Bruckmann
Delphi Automotive Systems, Luxembourg
{thomas.bruckmann}@delphi.com

ABSTRACT

We present CoCoTest, a tool for automated testing of continuous controllers at the Model-in-the-Loop stage. CoCoTest combines explorative and exploitative search algorithms to identify scenarios in the controller input space that violate or are likely to violate the controller requirements. This enables a scalable and systematic way to test continuous properties of such controllers. Our experiments show that CoCoTest identifies critical flaws in the controller design that are rarely found by manual testing and go unnoticed until late stages of embedded software system development.

Categories and Subject Descriptors [Software Engineering]: Testing and Debugging

Keywords: Search-based testing; continuous controllers; automotive software systems; MATLAB/Simulink models.

1. INTRODUCTION

The number and the complexity of software components embedded in today's vehicles is rapidly increasing. A large group of these components monitor and control the operating conditions of physical low-level devices, e.g., components controlling the velocity of a DC motor or the position of a flap. These controllers are known as *continuous controllers* and comprise more than half of the controllers used in industry [5]. To identify early design errors of continuous controllers, engineers create a model of the environment, capturing the behavior of the device that interacts with a controller, and perform testing and simulations of the controller and the environment models. This stage of testing is known as *Model-in-the-Loop (MiL)* [9] testing and is performed in various embedded system sectors such as the automotive domain. The subsequent stages of controller development are referred to as *Software-in-the-Loop (SiL)* and *Hardware-in-the-Loop (HiL)* [9]. Compared to SiL and HiL, the development and testing at MiL level are considerably faster as the engineers can quickly modify the controller model and immediately test the system. In addition, MiL testing is much less expensive than SiL or HiL testing.

Testing continuous aspects of control systems is challenging and is not yet supported by existing tools and techniques [2, 9]. Many existing approaches to testing embedded software systems focus on analyzing discrete or mixed discrete-continuous systems [2, 7]. These techniques, however, are not amenable to analyzing controllers with pure continuous behavior. Search-based techniques have been applied to Simulink models for generation of complex input signals and test input data for Simulink models [1]. However, they do not address the generation of test cases with respect to system requirements and do not provide any insight as to how one can develop test oracles for the generated test inputs. Finally, a number of commercial verification and testing tools have been developed, aiming to generate test cases for Simulink models, namely the Simulink Design Verifier software [8], and Reactis Tester [6]. Currently, these tools handle only combinatorial and logical blocks of the Simulink models, and fail to generate test cases that specifically evaluate continuous blocks (e.g., integrals or PIDs) [9].

In our earlier work, we proposed a search-based approach to MiL testing of continuous controllers [3, 4]. We identified and formalized a set of requirements for continuous controllers, and developed a search-based technique to generate test cases with respect to these requirements. Our proposed technique combines *explorative* and *exploitative* search algorithms [4] to compute scenarios in the controller input space that violate or are likely to violate the controller requirements. In this paper, we present **CoCoTest** (**Co**ntinuous **Co**ntroller **Test**er) that supports our automated approach to MiL testing of continuous controllers [3, 4]. In addition, CoCoTest provides a full-fledged user interface, allowing engineers to create and manipulate test configurations, tune search algorithms, monitor and examine the resulting *HeatMap* diagrams [3] visualising the search input space, and apply MiL testing for any desired input within the controller search space. Our experiments show that CoCoTest is effective and generates test cases violating requirements and that had not been previously found by manual testing based on domain expertise [4]. CoCoTest is developed in the context of our research-based collaboration with Delphi in Luxembourg, a leading global supplier of electronics and technologies to automotive companies.

2. BACKGROUND AND MOTIVATION

In this section, we provide background on MiL testing of continuous controllers, and motivate CoCoTest using the needs of the automotive domain. Figure 1(a) shows an overview of a controller and a plant (environment) model in a feedback loop. The system input and output are respectively shown as *desired* and *actual* in Figure 1(a). Input (*desired*) represents the position we need a valve (plant) to arrive at, for instance, and output (*actual*) represents the actual position of the valve (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 or practically negligible. The task of the controller is to eliminate the error by manipulating the plant to obtain the desired effect on the actual status of the plant.

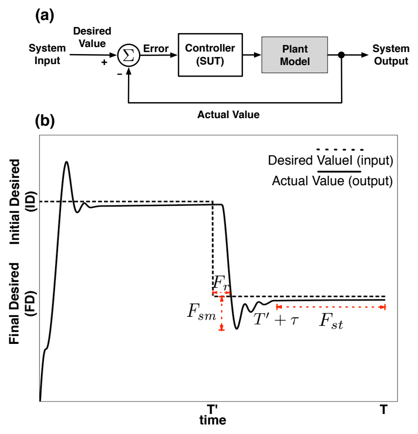


Figure 1: Continuous Controllers: (a) Overview and (b) Input/output diagrams.

Continuous controllers are typically specified in Simulink. Figure 1(b) shows simulations representing the input and output of a controller. The input, i.e., *desired*, is shown by a dashed line and is given as a step signal. Specifically, this input signal first sets the controller at an *initial desired (ID)* value until time T', and then requires the controller to move to a *final desired (FD)* value by time T. The output, i.e., *actual*, shown by a solid line, starts at zero, and gradually moves to reach and stabilize at the *initial desired* (ID), and then it moves towards the *final desired* (FD) and stabilizes there.

To test a controller, engineers simulate the controller using different input step signals by varying ID and FD. For each simulation, they generate the output signal, i.e., the *actual* signal in Figure 1(b), and check if the output conforms to the following three main requirements that we identified in our previous work [3, 4]: *Stability:* The controller shall guarantee that the output will reach and stabilize at the input after a time limit. *Smoothness:* The actual value shall not change abruptly when it is close to the input. *Responsiveness:* The controller shall respond within a time limit.

We define three objective functions F_{st}, F_{sm} and F_r over the output signal to estimate quantitative values for stability, smoothness and responsiveness requirements, respectively. We provided formal definitions of these functions in our earlier work [3, 4]. Briefly, F_{st} (stability) measures the maximum difference between input and output over the time periods shown by thin dashed arrows in Figure 1(b). Function F_{sm} (smoothness) measures the maximum undershoot and overshoot of the output signal. Function F_r (responsiveness) measures the response time intervals of the output as shown in Figure 1(b).

We provide an input to the controller Simulink model, and use the generated output signal to compute F_{st}, F_{sm} and F_r. The input to the controller includes values for ID and FD, which characterize the input step signal. Having computed these three functions over an output signal, engineers can then decide, based on their domain knowledge and thresholds provided in the requirements, whether the controller under analysis satisfies each of the controller requirements or not. The higher the objective function value, the more likely it is that the controller violates the requirement corresponding to that objective function.

In our earlier work, we proposed a search-based approach for testing continuous controllers [3, 4]. We developed search algorithms maximizing F_{st}, F_{sm} and F_r within an input search space with two dimensions: ID and FD. For example, consider a controller with float variables ID and FD ranging from 0 to 1. The input search space of this controller with dimensions ID and FD is

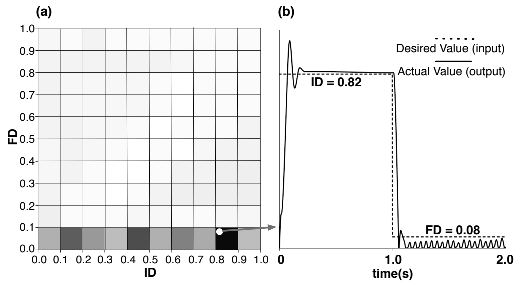


Figure 2: An example representing application of CoCoTest to a faulty controller: (a) a HeatMap diagram, and (b) a scenario violating the stability requirement.

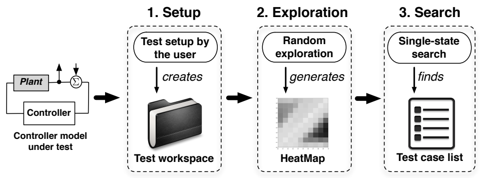


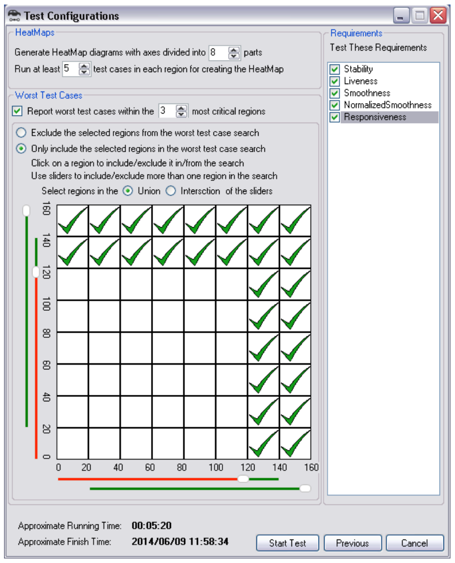
Figure 3: An overview of the MiL testing approach for controllers implemented in CoCoTest.

shown in Figure 2(a). Our approach has two steps: exploration and search. For each objective function, our approach first computes that function for points randomly selected from the input space (exploration step). We divide the input space into a number of regions, e.g., 100 equal regions in Figure 2(a), and shade each region based on the objective function output. The resulting diagram is called a *HeatMap* diagram. For example, Figure 2(a) is a HeatMap diagram generated based on the stability objective function values for 1000 points. Specifically, in a HeatMap diagram, the points in darker regions yield a higher average objective function output than the points in lighter regions. Hence, darker regions are more likely to include input values violating the controller requirements. In the search step, we apply a single-state search algorithm to selected dark regions entailing higher risks, e.g., of damaging the controlled device, to find points that maximize our objective functions, and hence, are more likely to violate the requirements.

For example, the HeatMap in Figure 2(a) is generated by CoCoTest based on the output of the stability objective function F_{st} applied to a faulty controller. The controller satisfies the stability requirement for all the input signals related to the points in the clear-shaded regions. However, applying CoCoTest to the dark region of (ID = $[0.8..0.9]$ and FD=$[0..0.1]$) in Figure 2(a) results in finding the simulation in Figure 2(b) which clearly violates the stability requirement. Note that since the controller behavior for most of the input space (more than 95%) conforms to the stability requirement, it is very unlikely that one can discover the faulty behavior by manually selecting and running a few simulations.

3. TOOL OVERVIEW

Figure 3 shows an overview of CoCoTest that consists of three main steps: (1) Creating and initializing a *test workspace*, i.e., a container for the test inputs and configuration data, by the user (setup), (2) exploring the input space of the controller and generating HeatMap diagrams (exploration), and (3) identifying the critical regions of the input space and computing the worst-case test scenarios in those regions (search). Critical regions are those that are more likely to include test scenarios violating the controller requirements, e.g., boundary regions are typically more critical since controllers are more likely to damage devices at boundary regions. CoCoTest offers end-to-end support for the process in Figure 3. Below, we discuss the three steps of the process in Figure 3.



Test Configurations

HeatMaps

Generate HeatMap diagrams with axes divided into 8 parts

Run at least 5 test cases in each region for creating the HeatMap

Worst Test Cases

✓ Report worst test cases within the 3 most critical regions

Exclude the selected regions from the worst test case search [✓ Only include the selected regions in the worst test case search]

Click on a region to include/exclude it in/from the search

Use sliders to include/exclude more than one region in the search

Select regions in the [✓ Union] Intersction of the sliders

Requirements

Test These Requirements

✓ Stability

✓ Liveness

✓ Smoothness

✓ NormalizedSmoothness

✓ Responsiveness

Approximate Running Time: **00:05:20**

Approximate Finish Time: **2014/06/09 11:58:34**

Figure 4: Configuring HeatMap diagrams in CoCoTest.

3.1 Setup

In the Setup step, the user creates a test workspace, which is a structure used to store the input data, HeatMap configurations, and the test results. CoCoTest provides the user with a wizard consisting of the following three steps to create a test workspace: (1) First, the user selects the Simulink file (*.mdl file) of the controller-plant model. (2) Second, the user provides the simulation time (T in Figure 1), the model variables corresponding to ID and FD variables, and the ranges of these variables. (3) Third, the user configures the layout and the setting of the HeatMap diagrams used to visualize the exploration results. Specifically, the user provides (i) the number of regions in a HeatMap, (ii) the minimum number of test cases to be generated in each region, and (iii) the critical operating regions of the controller.

Figure 4 shows the last step of the wizard for a DC motor controller [4]. For this example, the value ranges of ID and FD are between 0 and 160, resulting in the HeatMap layout in Figure 4. The user, then, chooses to divide the diagram into 64 regions, and in addition, specifies that at least five test cases should be generated and executed in each region of the diagram. Then, the user specifies the critical regions by clicking on the appropriate HeatMap regions. These selected critical regions are candidate inputs regions for the search step (step 3). For example in Figure 4, the regions where either the initial or the final DC motor speed is between 120 and 160 are selected as the critical operating regions of the controller. This is because the controller is likely to damage the DC motor when the motor speed is at least 75% of its maximum speed, or when the motor is accelerating to reach at least 75% of its maximum speed. Finally, the user can specify the number of worst-case scenarios to be generated per requirement, e.g. three in Figure 4.

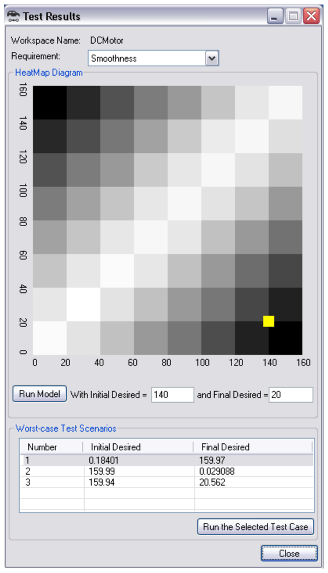
CoCoTest estimates the running time of the test (exploration and search steps) based on the given data, e.g., five minutes and 20s in Figure 4. CoCoTest can also be executed in a maintenance mode, allowing an advanced user to choose search algorithms for the exploration and search steps and to configure the specific parameters of these algorithms.

3.2 Exploration

During exploration, CoCoTest applies an adaptive random (unguided) search to the entire input space in order to identify the high risk areas. The search selects inputs in the search space randomly in order to provide an unbiased estimate of the average objective function values in different regions. These regions and the computed averages are then visualized using HeatMap diagrams. CoCoTest provides two options for explorative search algorithm: random search and adaptive random search [4].

The overall structure of both random and adaptive random search algorithms are implemented as MATLAB script *templates* with some placeholders in CoCoTest. These templates are then instantiated and populated using the test workspace created by the user in the setup step. For example, the number of iterations of the search is a placeholder that has to be filled based on the number of the test cases that the user wishes to generate and execute in each region.

The exploration step produces one HeatMap diagram for each controller requirement (see Figure 5). CoCoTest allows the user to select a requirement from a drop down list to retrieve the HeatMap diagram corresponding to that requirement. In addition, the user can run any arbitrary test case by simply double clicking on a point on the HeatMap, or entering the ID and FD values and pushing the *Run Model* button (see Figure 5). For example in Figure 5, pushing the *Run Model* button causes the test case with ID=140 and FD=20 to be executed. A yellow rectangle indicates the position of the selected test case on the HeatMap diagram.



Test Results

Workspace Name: DCMotor

Requirement: Smoothness

HeatMap Diagram

With Initial Desired = 140 and Final Desired = 20

Worst-case Test Scenarios

Number	Initial Desired	Final Desired
1	0.18401	159.97
2	159.99	0.029088
3	159.94	20.562

Figure 5: Test results form in CoCoTest.

3.3 Search

During the search step, CoCoTest applies a heuristic single-state search to a few selected regions in order to find worst-case test scenarios that are likely to violate the controller requirements. Given a HeatMap region, the search starts with the point with the highest objective function value among those computed during exploration, and then, it iteratively generates new points by tweaking the current point. The specific tweak operators used in our search are discussed in [4]. After a number of iterations, it reports the point with the worst (highest) objective function value as the worst-case test scenario in the given region. In the maintenance mode, one can choose and configure one of the following single-state search algorithms: Hill-Climbing, Hill-Climbing with Random Restarts and Simulated Annealing algorithms for single-state search [4].

Similar to the exploration algorithm, different single-state search algorithms are implemented as MATLAB script templates. For each HeatMap region, CoCoTest instantiates the search algorithm template file with the given values during the setup step and the boundaries of the critical region. It then calls MATLAB to run the instantiated search script and computes the worst-case test scenario in the region. The output of the search step is a list of worst-case test scenarios for each controller requirement. By selecting a requirement from the drop down list at the top in Figure 5, the list of worst-case test scenarios corresponding to that requirement appears in the list at the bottom of that figure. Each test case is characterised by the ID and FD values, e.g., ID=159.94 and FD=20.562 for the third generated test case in Figure 5. The user can rerun each of these test cases to view simulation results by double clicking on the test cases in the list.

4. EVALUATION

We evaluated the practical utility of CoCoTest by applying it to a representative industrial case study from automotive domain (with 443 Simulink blocks) and to a publicly available controller [1] (with 21 Simulink blocks) [3, 4]. Our experiments show that CoCoTest automatically generates several test cases for which the MiL level simulations indicate potential errors in the controller model. Furthermore, the resulting test cases had not been previously found by manual testing based on domain expertise. For example, regarding smoothness, CoCoTest found a test scenario with an undershoot around %20 for the industrial controller. The maximum undershoot/overshoot for the same controller identified by manual testing was around %5. Similarly, for the responsiveness property, CoCoTest 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.

We have made CoCoTest available to Delphi, our partner company, and have presented it in a hands-on tutorial to eight Delphi engineers. Post training, the engineers were able to independently use CoCoTest. We received positive feedback after the tutorial. According to the engineers, CoCoTest is particularly useful at early stages of controller design to identify and detect design flaws. They noted that the HeatMap diagrams, in addition to enabling the identification of critical regions, can be used in the following ways: (1) They can gain confidence about the controller behaviors over the light shaded regions of the diagrams. (2) The diagrams enable them 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 clear to dark. A sharp contrast such as a dark region neighboring a light-shaded region may potentially indicate an abnormal behavior that needs to be further investigated.

To further evaluate CoCoTest, we seeded 3 faults into the industrial controller model, and used CoCoTest to find those faults. The faults were provided by Delphi engineers and were chosen based on years of debugging experience. For all these faulty models, CoCoTest was able to find test cases violating at least one of the requirements. For example, the HeatMap and test scenario in Figure 2, representing a test case violating the stability requirement, are obtained by applying CoCoTest to a fault-seeded controller.

5. IMPLEMENTATION

Figure 6 shows the architectural view of CoCoTest (`https://sites.google.com/site/cocotesttool/`). CoCoTest adopts a three-tier architecture with an extra service layer which handles the configuration management. The exploration and search algorithms are implemented in MATLAB script templates as described in section 3. The templates execute model simulations to compute the objective function values. CoCoTest calls MATLAB from command

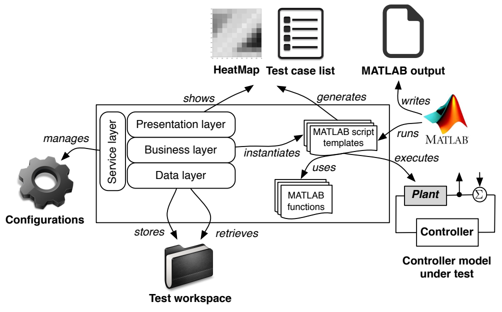


Figure 6: CoCoTest architectural view.

line to run the MATLAB scripts. It redirects MATLAB output to a file and periodically reads the file to know when the test execution is finished and the test results are ready.

CoCoTest is implemented in Microsoft Visual Studio 2010 and Microsoft .NET 4.0. It is an object-oriented program in Visual C# with 65 classes and roughly 30K lines of code excluding comments. The main functionalities of CoCoTest have been tested with a test suite containing 200 test cases. CoCoTest requires 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. MATLAB 2007 was selected due to compatibility with Delphi models.

6. CONCLUSION

We presented a tool, CoCoTest, to automate MiL testing of continuous controllers. The tool enables users to setup test workspaces, explore and visualise the controller input search space, select the critical operating regions of controllers, and identify worst-case test scenarios in the selected regions. Our evaluation on realistic case studies indicates that CoCoTest is able to generate test cases for which the MiL level simulations indicate potential errors in the controller model. Furthermore, the test cases generated by CoCoTest had not been previously found by manual testing based on domain expertise. CoCoTest is implemented as part of an industry-driven research effort to devise cost-effective and scalable techniques to test automotive software components.

Acknowledgments

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

7. REFERENCES

[1] F. Elberzhager, A. Rosbach, and T. Bauer. Analysis and testing of Matlab Simulink models: A systematic mapping study. In *JAMAICA 2013*. ACM, 2013.

[2] T. Henzinger and J. Sifakis. The embedded systems design challenge. In *FM*, pages 1–15, 2006.

[3] R. Matinnejad, S. Nejati, L. Briand, T. Bruckmann, and C. Poull. Automated model-in-the-loop testing of continuous controllers using search. In *Search Based Software Engineering*, pages 141–157. Springer, 2013.

[4] R. Matinnejad, S. Nejati, L. Briand, T. Bruckmann, and C. Poull. Search-based automated testing of continuous controllers: Framework, tool support, and case studies. *Information and Software Technology*, 2014.

[5] N. S. Nise. *Control Systems Engineering*. John-Wiely Sons, 4th edition, 2004.

[6] Reactive Systems Inc. Reactis Tester. `http://www.reactive-systems.com/simulink-testing-validation.html`, 2010.

[7] T. Stauner. Properties of hybrid systems-a computer science perspective. *Formal Methods in System Design*, 24(3):223–259, 2004.

[8] The MathWorks Inc. Simulink. `http://www.mathworks.nl/products/simulink`, 2003.

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

[1] `http://www.mathworks.com/matlabcentral/fileexchange/11587-dac-motor-model-simulink`