Demo: Fuzzing Cyber-Physical System Development Environments With CyFuzz

Shafiu Azam Chowdhury
The University of Texas at Arlington
Arlington, Texas 76010
shafiuazam.chowdhury@mavs.utd.edu

Taylor T. Johnson
Vanderbilt University
Nashville, Tennessee 37240
taylor.johnson@vanderbilt.edu

Christoph Csallner
The University of Texas at Arlington
Arlington, Texas 76010
csallner@uta.edu

ABSTRACT

Hardening cyber-physical system (CPS) development environments by finding bugs is vital as these tool chains generate artifacts that are deployed in safety-critical environments. In this demonstration we present a prototype implementation of CyFuzz, which is the first differential testing framework for CPS development environments. CyFuzz currently targets the popular Simulink tool chain. CyFuzz automatically generates random, but valid Simulink models and uses them to test Simulink, by varying compilation and simulation options and looking for result discrepancies between these simulations and executions. Our automated tool has generated thousands of valid Simulink models to date, among others, have semi-independently reproduced a confirmed bug in Simulink (version R2015a) and identified interesting issues in the popular CPS development tool chain.

 ACM Reference format:
DOI: 10.1145/nnnnnn.nnnnnn

1 INTRODUCTION

Cyber-physical system (CPS) development environments are widely used in industry. Typical tool chains contain simulators, compilers, and code generators. Using a sophisticated development environment such as MathWorks’ Simulink, engineers can design, simulate, and test their systems and generate code for production environments [9]. Reducing the number of bugs present in CPS development environments is crucial, as bugs in the tool chain may introduce unintended behavior in simulations and generated code.

While it would be ideal to formally verify that an entire CPS development tool chain is bug-free, unfortunately this is practically infeasible. Moreover it is often not possible to get a full, up-to-date, formal specification of a commercial CPS tool chain [8]. In contrast, differential testing does not need full formal specifications, as it compares the results of two executions or simulations that are supposed to produce the same results. Differential testing has been effective in recent compiler testing projects; collectively finding hundreds of bugs in commercial compiler implementations that are part of CPS development tool chains [3, 4, 6, 10]. Our recent study of publicly available Simulink bug-reports also suggests differential testing as a good candidate for finding CPS tool-chain bugs [2].

CyFuzz is, to the best of our knowledge, the first differential testing framework for CPS development environments. CyFuzz generates random (“fuzz”), but valid CPS models and tests the system under test (SUT) using differential testing—on various SUT configurations [2]. While existing work mostly focuses on finding bugs in Simulink models [1, 5], CyFuzz targets the CPS tool chain [2]. Other work targeting CPS tool chain components requires a formal SUT specification, which is often not available [7, 8].

In this demonstration, we show how to setup CyFuzz, use CyFuzz to generate valid Simulink models, and finally use the models for differential testing of Simulink. The CyFuzz source code and current evaluation results are available at the CyFuzz homepage.

2 A DIFFERENTIAL TESTING FRAMEWORK FOR CPS TOOL CHAINS

CyFuzz supports the conceptual CPS modeling framework commonly found in CPS tool chains such as Simulink. Specifically, Simulink’s models follow the data-flow paradigm and may contain individual procedural blocks.

At a high level, CyFuzz has two subcomponents: a random model generator and a comparison framework. The generator automatically creates random CPS models based on the configuration options set by the user. Options include the number of blocks in each model, the depth of the model hierarchy, and the probability of picking a block from a given library. The generator’s first phase chooses blocks randomly (according to the probabilities) and places them in an empty model and configures some block parameters with random values. The second phase arbitrarily chooses and connects block-ports, defining the model’s data-flow. The resulting model may be rejected by the SUT’s compiler. CyFuzz tries to fix such errors, by changing the model iteratively in a feedback-driven model generation approach [2].

After generating a valid model, CyFuzz passes it to the comparison framework, which simulates the model many times under varying user-defined SUT configuration options. CyFuzz logs signal data at each simulation step and compares them, recording any dissimilarity in block-output data at the final simulation step for further manual inspection.

1 Available: https://github.com/verivist/lsf_ranogen
We demonstrate each component of CyFuzz. CyFuzz is fully auto-
we semi-independently discovered one existing bug. Figure 1 is a
work only varies various simulation modes (e.g., Accelerator Mode
via its S-function interface. CyFuzz leverages Csmith [10] to
generate random, but well-defined C code (according to the C99
standard) and uses them in the models as s-functions, hence
adding procedural code inside individual data-flow nodes.

The current prototype implementation of the comparison frame-
work only varies various simulation modes (e.g., Normal Mode,
Accelerator Mode, and Rapid Accelerator Mode) and toggles
simulation optimization [9]. Extended implementation details are
available elsewhere [2].

3 CYFUZZ PROTOTYPE FOR SIMULINK

Our prototype chooses blocks from four built-in Simulink libraries:
Continuous, Discrete, Sink, and Sources. The prototype fixes
different types of errors, including algebraic loops and data-type
incompatibilities between blocks. The tool can generate hierar-
chical models using subsystem and for-each blocks and can log
simulation results using various Simulink APIs.

Besides using built-in blocks, Simulink can define custom block-
behavior by placing procedural code (e.g., C or Matlab code) directly
via its S-function interface. CyFuzz leverages Csmith [10] to
generate random, but well-defined C code (according to the C99
standard) and uses them in the models as s-functions, hence
adding procedural code inside individual data-flow nodes.

In these experiments more than 79% of the generated models
could be compiled and simulated successfully. The bottleneck of Cy-
Fuzz’s implementation was the Log Signals phase, in which Simulink
simulates the models using different configuration options, which
indeed is time consuming. However, the overall runtime (some
52 seconds on average) for completing a single experiment seems
acceptable.

In our experiments we did not find any new bugs, however,
we semi-independently discovered one existing bug. Figure 1 is a
screen-shot of the top level of the CyFuzz-generated model that
exposes this bug [2].

5 TOOL DETAILS AND DEMONSTRATION

We demonstrate each component of CyFuzz. CyFuzz is fully auto-
mated, to continuously generate random models and run them in
the comparison framework. Its command-line interface parses a
configuration file supplied by the user. CyFuzz is mostly written
in Matlab and supports parallel execution of the framework by
creating multiple instances of the project. CyFuzz does not depend
on any other tool except a customized version of Csmith [10] for
generating random C code for s-functions. To date, CyFuzz sup-
ports Simulink 2015a only; we have not tested CyFuzz in recent
versions of Simulink.

While running experiments, CyFuzz stores generated models and
comparison results in the file system. The user can interpret the
results using a Matlab script. If the script reports any comparison
errors the user can investigate the comparison results and inspect
the associated model manually. CyFuzz also has a Python script to
detect Matlab crashes which can be useful for starting the exper-
iment automatically and to investigating the crashes later. CyFuzz’s
user manual is available on the project homepage.

To demonstrate the generator phases, we will instruct CyFuzz
to generate models in an interactive fashion, pausing after each
phase of the generator and the comparison framework and high-
lighting the core functionality of that phase. As an example, we
will demonstrate how CyFuzz iteratively fixes several errors from
a randomly generated model and will visually present simulating
and comparing outputs in the comparison framework.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science
Foundation under Grants No. 1446311 and 1527398, by Air Force
Office of Scientific Research (AFOSR) contract numbers FA9550-
15-1-0258 and FA9550-16-1-0246, and by Air Force Research Lab
(AFRL) contract number FA8750-15-1-0105. Any opinions, find-
ings, and conclusions or recommendations expressed in this publica-
tion are those of the authors and do not necessarily reflect the views of
AFRL, AFOSR, or NSF.

REFERENCES

analysis for improving simulation coverage of Simulink/Stateflow models. In
Proc. 8th ACM & IEEE International Conference on Embedded Software (EDSOFT).
ACM, 89–98. DOI: http://dx.doi.org/10.1145/1450058.1450071
[2] Shafiu Azam Chowdhury, Taylor T. Johnson, and Christoph Csallner. 2016. Cy-
Fuzz: A differential testing framework for cyber-physical systems development
environments. In Proc. 6th Workshop on Design, Modeling and Evaluation of Cyber
Physical Systems (CyPhy) Springer International Publishing, Cham, 46–60. DOI:
http://dx.doi.org/10.1007/978-3-319-51738-4_4
[3] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust Type-
checker Using CLP (T). In Proc. 30th IEEE/ACM International Conference on
1109/ASE.2015.65
presentation/holler
SimCoTest: A test suite generation tool for Simulink/Stateflow controllers. In
DOI: http://dx.doi.org/10.1145/2889160.2889162
08/02/introducing-jsfunfuzz/(2007).
Time and Embedded Technology and Applications Symposium. IEEE, 205–214. DOI:
http://dx.doi.org/10.1109/RTAS.2007.39
[8] Ingo Stümer, Mirko Conrad, Heiko Dörz, and Peter Pepper. 2007. Systematic Test-
ing of Model-Based Code Generators. IEEE Transactions on Software Engineering
(TSE) 33, 9 (Sept. 2007), 622–634. DOI: http://dx.doi.org/10.1109/TSE.2007.70708
com/help/simulink/(2017).
derstanding bugs in C compilers. In Proc. 32nd ACM SIGPLAN Conference on
Programming Language Design and Implementation (PLDI). ACM, 283–294. DOI:
http://dx.doi.org/10.1145/1993498.1995332