Understanding and Improving Cyber-Physical System Models and Development Tools

Shafiul Azam Chowdhury
University of Texas at Arlington
Arlington, Texas
shafiulazam.chowdhury@mavs.uta.edu

ABSTRACT

Recent years have seen an increasing interest in understanding and analyzing cyber-physical system (CPS) models and their development tools. Existing work in this area is limited by the lack of an open corpus of CPS models, which we aim to address by building the by-far largest curated corpus of CPS artifacts. Next, to address the safety-critical aspect of CPS development tools, we discuss the design and evaluation of the very first differential testing framework for arbitrary CPS tool chain. We identify challenges unique to commercial CPS tool chain testing and present a tool implementation which has already found 9 new, confirmed bugs in Simulink, the most widely used CPS development tool.

CCS CONCEPTS

• **Software and its engineering → Model-driven software engineering**; **Software testing and debugging**;

KEYWORDS

Cyber-physical systems, differential testing, Simulink

ACM Reference Format:
Shafiul Azam Chowdhury. 2018. Understanding and Improving Cyber-Physical System Models and Development Tools. In *ICSE '18 Companion: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3183440.3183455

1 INTRODUCTION AND MOTIVATION

In model-based software engineering, CPS designers rapidly prototype their systems using graphical models and generate executable code for the target hardware, using sophisticated development tools. As most CPS operate in safety-critical environments, it is crucial to remove bugs from the development tools which can compromise the fidelity of the generated CPS artifacts.

Formal verification of commercial CPS tool chains (e.g. MathWorks' Simulink) does not yet scale to their millions of lines of code [1, 5]. Recent advances in automated testing, on the other hand, have collectively found over 1,000 bugs in widely used compilation tools. However, when testing CPS tool chains we face additional challenges since CPS modeling languages differ significantly from traditional programming languages [1].

Next, increased usage of model-based development has elicited interest in understanding various model metrics (e.g., size measures) and model-based software engineering research. However, when evaluating the proposed metrics, most of the model-based studies only use a handful of models, which could adversely affect the evaluation. Since a model corpus is currently publicly unavailable, insights into modeling practices are also currently unknown.

In this series of works, we will develop and maintain the first and largest-ever publicly-available corpus of CPS models. Initial results suggest the utility of the corpus in revealing interesting model-based software engineering practices, insights on model metrics and benefits in tool development and empirical research. Next, we discuss the design, implementation, and evaluation of a testing framework for commercial CPS tool chains, through addressing challenges unique to CPS tool testing and exploring schemes to address the absence of formal tool chain specifications.

2 CPS MODEL CORPUS AND METRICS

Partly to facilitate experimental evaluation of the tools that will be developed in this project, we are currently developing the first large-scale corpus of freely available, open-source CPS artifacts (such as models and integrated code and binaries), for mainstream commercial CPS tool chains. We will investigate various model metrics, including the complexity of the CPS models and code. We will also analyze industrial models and publish the metrics data only where distributing the artifacts would be prohibited. The corpus infrastructure will be freely available to others, to ease evaluation and reduce the non-trivial artifact collection overhead in other research.

3 CPS TOOL CHAIN TESTING

Here, we introduce *CyFuzz* — the first known conceptual differential testing framework for CPS tool chain. Next, we present various improvements over it which led to finding new bugs in Simulink.

3.0.1 A Conceptual Differential Testing Framework. CyFuzz's state-of-the-art model generator creates random CPS models based on the configuration options set by the user. After creating a possibly erroneous model, in the Fix Errors phase, the generator repeatedly attempts to fix any compilation or runtime exceptions in it. CyFuzz simulates a valid model many times under varying user-defined simulation modes and compiler optimization levels. CyFuzz logs signal data at each simulation step and compares them, recording any dissimilarity in block-output data which could potentially indicate a bug [1].

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3183455

3.0.2 Incorporating Specifications in Model Generation. One limitation of the feedback-driven error fixing approach is that it requires several time-consuming iterations to fix the incorrectly generated models and do not leverage any of the tool chain specifications, which are often available publicly. Furthermore, the simple, heuristic-based approach may not even terminate, especially when the models are large. To circumvent, SLforge (our recent improvement over CyFuzz) uses regular expression-based parsers to automatically collect Simulink specifications which are available publicly in natural language [2].

3.0.3 Expressive Models. Earlier works attribute expressive, feature-rich model generation to effective bug discovery [4, 6]. CyFuzz-generated models lack many essential modeling constructs (i.e. conditional executions, loops, bit and logic operations, subsystems and model referencing and math operations) and hence are not expressive enough to find new bugs. To circumvent, SLforge incorporated these features in the generated models and have already found 12 confirmed bugs in the Simulink tool chain. The tool can generate large models with complex hierarchies and automatically incorporate native C code using a customized version of Csmith [6].

3.0.4 EMI-testing. Equivalent modulo input (EMI), a recent advancement in differential testing, systematically mutates a source program as long as its semantics remain equivalent under the given input data [4]. SLforge currently statically mutates a randomly generated model by pruning some of the *dead* blocks discovered in a graph-traversal on our intermediate model-representation. Within this context, blocks which are not connected to output-producing *sinks* do not participate in the model execution, and are considered as dead blocks [5]. We will investigate *dynamically* detecting dead components in a model through computing model component coverage and exploring other EMI techniques besides pruning [4].

3.0.5 Bug Classification and Test Reduction. Bugs which *silently* incur wrong runtime-output are considered more severe than crashes. We will prioritize finding these bugs by targeting the most critical *Embedded Coder* in Simulink and will automatically classify the discovered bugs into various categories, based on their discovery-point (i.e. compile-time, runtime, EMI-variant) and root-cause.

Currently, before reporting a bug to the tool chain manufacturer, we manually reduce the respective (potentially large) models to smaller versions which can still reproduce the bug. Since model reduction is essential as they aid their comprehension and saves valuable investigation time, we will incorporate automated test reduction techniques in our framework.

3.0.6 Specification Inference. Besides collecting specifications through documentation parsing, we will automatically gather formal tool chain semantics. While different variants of *dynamic symbolic execution* to describe and infer formal specifications have been discussed in the literature, we will explore and evaluate these schemes to appropriately capture CPS development tool semantics, for the very first time [3].

To drastically simplify the huge challenge of analyzing an entire CPS development tool semantics, we will instead systematically split this task into much smaller and therefore more manageable sub-tasks, by extracting semantics from smaller subsets of the tool. However, this approach will reduce our coverage of extracted block semantics by targeting only that portion of a CPS development tool for which automatic code-generation support is available. Nevertheless, we find the technique useful as we obtain semantics for a reasonably large portion of some CPS development tool and widely used simulation solvers and code-generators. We will leverage the corpus developed in §2 to prioritize the analysis of blocks and model configuration settings.

3.1 Initial Results

Our realization to test the Simulink tool chain generates large hierarchical models and varies simulation modes to trigger code generation and toggles optimization. Initial results are promising: we have found and reported 14 unique bugs to MathWorks to date, out of which 9 have been confirmed as new bugs and 3 have been confirmed as reproduction of known issues.

We have also evaluated performance and bug-finding capabilities of different analysis techniques, i.e. Fix Errors heuristics [1] and enabling and disabling the specification usage [2]. Perhaps not surprisingly, when using specifications in model generation, the tool both found more bugs and performed better in terms of runtime [2].

ACKNOWLEDGMENTS

The material presented in this paper is based upon work supported by the National Science Foundation (NSF) under grant numbers CNS 1464311, CNS 1713253, EPCN 1509804, SHF 1527398, and SHF 1736323, the Air Force Research Laboratory (AFRL) through the AFRL's Visiting Faculty Research Program (VFRP) under contract number FA8750-13-2-0115, as well as contract numbers FA8750-15-1-0105, and FA8650-12-3-7255 via subcontract number WBSC 7255 SOI VU 0001, and the Air Force Office of Scientific Research (AFOSR) through AFOSR's Summer Faculty Fellowship Program (SFFP) under contract number FA9550-15-F-0001, as well as under contract numbers FA9550-15-1-0258 and FA9550-16-1-0246. The U.S. government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of AFRL, AFOSR, or NSF.

REFERENCES

[1] Shafiul Azam Chowdhury, Taylor T. Johnson, and Christoph Csallner. 2016. CyFuzz: 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.

[2] Shafiul Azam Chowdhury, Soumik Mohian, Sidharth Mehra, Siddhant Gawsane, Taylor T. Johnson, and Christoph Csallner. 2018. Automatically Finding Bugs in a Commercial Cyber-Physical System Development Tool Chain With SLforge. In *Proc. 40th International Conference on Software Engineering (ICSE)*. ACM.

[3] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. 2008. DySy: Dynamic Symbolic Execution for Invariant Inference. In *Proceedings of the 30th International Conference on Software Engineering*. ACM, New York, NY, USA.

[4] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 216–226.

[5] The MathWorks Inc. 2017. Simulink Documentation. http://www.mathworks.com/help/simulink/. (2017). Accessed Oct. 2017.

[6] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 283–294.