Automatically Finding Bugs in Commercial Cyber-Physical System Development Tool Chains

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

ABSTRACT

Commercial Cyber-physical System (CPS) development tools (e.g. MathWorks' Simulink) are widely used to design, simulate and automatically generate artifacts which are deployed in safety-critical embedded hardware. CyFuzz, the state-of-the-art CPS tool chain testing scheme is inefficient, cannot generate feature-rich inputs and is ineffective in finding new tool chain bugs. To better understand various properties of publicly available CPS models, we conducted the first large-scale study of 391 publicly-available Simulink models. Next, we proposed an efficient CPS model-generation scheme capable of creating large, feature-rich random inputs. Our tool realization for testing Simulink which found 8 new confirmed bugs, along with the study-artifacts are publicly available.

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. Automatically Finding Bugs in Commercial Cyber-Physical System Development Tool Chains. In *ICSE '18 Companion: 40th International Conference on Software Engineering Companion, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3183440.3190330

1 INTRODUCTION AND MOTIVATION

Cyber-physical system developers heavily use complex development tools (e.g. MathWorks' Simulink) to design and verify graphical *models* and generate deployable artifacts. It is crucial to eliminate bugs from the complex tool chains as bugs may compromise the fidelity of the artifacts (i.e. code and binaries) they generate which are often deployed in safety-critical embedded hardware [1, 20].

Automated *differential testing* through feature-rich random test generation has been proven effective in collectively finding over thousands of production-grade compiler bugs [7, 27]. This inspired the *CyFuzz* project to identify challenges unique to the differential testing of CPS tool chains for the very first time and release a prototype implementation to automatically test the Simulink tool chain [3]. However, CyFuzz's heuristic-based approach to generate valid Simulink models, besides being runtime-inefficient, does not scale to large, hierarchical and feature-rich model generation. This largely inhibits CyFuzz's bug-finding capabilities and consequently, the scheme did not find any new bugs when testing the Simulink tool chain [2, 3].

Furthermore, unavailability of large-scale studies of popular Simulink modeling practices prohibited evaluation of CyFuzz's capability of generating *realistic* models. To circumvent, we conducted the first large-scale study of public Simulink models and observed metrics relevant to the generation of random models which have properties similar to the publicly-available models designed by researchers and engineers.

Here, we discuss an efficient Simulink model generation technique and present our experience with SLforge - an automated, open source tool which we have developed extending CyFuzz's code base [4]. Unlike CyFuzz, SLforge parses modeling specifications from official Simulink documentations which are described in semi-structured, natural language and performs various analyses to generate valid models by construction. We also discuss the first *equivalent modulo input (EMI)*-testing for CPS tool chain which has been proven recently as an effective compiler testing scheme [14]. Finally, we evaluate SLforge's efficiency, model generation, and bug-finding capabilities by comparing it with CyFuzz.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5663-3/18/05…$15.00
https://doi.org/10.1145/3183440.3190330

2 BACKGROUND AND RELATED WORK

Commercial CPS development tools enable users to design a CPS as a set of graphical dataflow *models*, which consist of *blocks*. A block accepts data through its *input ports*, typically performs on the data some operation, and may pass output to other blocks, along *connection* lines [4]. Tools typically offer hierarchical model creation through leveraging many *libraries* of built-in blocks, along with facilitating custom block-behavior by placing native code (e.g., C). Simulink-specific details are available [26].

Differential testing mechanically generates inputs and presents them to comparable variations of the software under test. Any execution variation for the same input likely indicates a bug [4, 17]. Differential testing of textual programming language compilers and analyses tools have been thoroughly investigated [5, 7, 12–15, 21, 27]. Existing works target parts of CPS tools based on unofficial and possibly outdated formal specifications [9, 22, 24, 25] and model transformation [18, 19]. Testing and analyzing the CPS models themselves [6, 8, 10, 11, 16, 23] are loosely related to our work.

CyFuzz, the state-of-the-art CPS tool chain-testing scheme automatically creates random CPS models based on the configuration

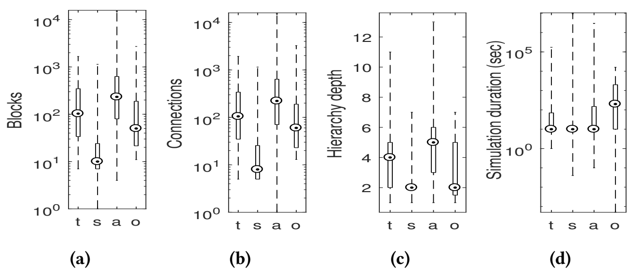

Figure 1: Collected models: Total blocks (a), connections (b), maximum hierarchy (c), and simulation duration (d) [4].

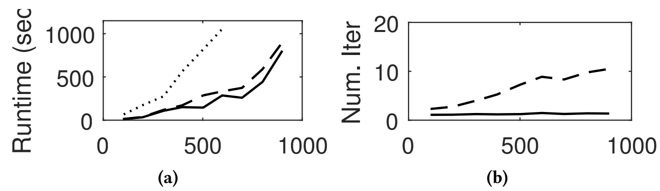

Figure 2: Runtime on valid models by model size given in blocks: (a) Average runtime of model generation; (b) Average number of required iterative fixes. Solid = SLforge; dashed = SLforge without specification usage and analyses; dotted = CyFuzz. [4]

options set by the user [3]. After creating a possibly invalid model, in the Fix Errors phase, CyFuzz repeatedly attempts to fix any specification violations in it. If succeeded, CyFuzz's comparison framework executes the model many times under varying user-defined Simulink configurations (i.e. simulation modes and compiler optimization levels) and logs execution data to compare them, recording any block-output dissimilarity which likely indicates a bug [3].

3 PUBLIC SIMULINK MODEL COLLECTION

To understand the properties of CPS data-flow models designed by both researchers and engineers, we conducted the first large study of 391 Simulink models available in various public sources [4]. Here we investigated model properties relevant to automated model generation, i.e. the number of blocks (connections) and hierarchy depth in models, distribution of blocks across hierarchy levels, and built-in libraries and custom block usage information. E.g., Figure 1 presents some of the metrics using min-max whisker plots, where we classified the models into the following groups: (1) *Tutorial* (*t*)—Simulink proprietary models; (2) *Simple* (*s*)—models we manually filtered out as toy-examples; (3) *Advanced* (*a*)—non-trivial models; (4) *Other* (*o*)—models from academic papers and search-engine results. The study results are available in details [4].

4 SLFORGE

Here we highlight the design of SLforge to address the various CyFuzz limitations (§1). First, we observed that CyFuzz does not leverage any of the available modeling specifications during model creation. Rather, it repeatedly compiles (runs) models to detect any modeling specification violation error through parsing tool chain reported error messages. Instead, we proposed generating valid models by construction leveraging automatically-collected modeling specifications. For the SLforge tool, we designed regular expression-based parsers to collect Simulink modeling specifications from the official web documentations. Once parsed, the specification *rules* are stored persistently using data-structures which can be reused by a parser for a different CPS tool using little engineering effort.

While creating a model in the tool-realization of SLforge, we use an intermediate representation (IR) for it and run various analyses to ensure that the generated model satisfies all of the collected rules. The analyses ensure that models are type-safe, satisfy the parsed specification rules, and contain no algebraic loops [26]. Any compilation error now would indicate a bug in the compiler implementation or a faulty specification.

Since collecting specifications for the entire Simulink tool chain would be overwhelming, we have added support for the widely used built-in libraries first, as directed by our public-model study. We also create EMI-variants from a model to identify *dead* blocks (i.e. blocks which does not have a directed path reaching a *Sink* block), and pruning them.

5 EVALUATION

We experimentally evaluated SLforge by comparing it with CyFuzz.

Efficient Model Creation. We generated 160 models in each of the following three experiments: two experiments used SLforge: (1) enabling specification usage and analyses and (2) disabling, and (3) in the third experiment, we used CyFuzz. Both SLforge versions had a lower average model-creation runtime than CyFuzz (Figure 2a), and when using specifications and analyses SLforge required much fewer (almost constant number of) iterations in the Fix Errors phase.

Feature-rich Model Generation. We experimentally verified that SLforge-generated models have similar size and hierarchy-depth properties when compared to the publicly available models whereas CyFuzz is incapable of generating large, hierarchical models [4]. Furthermore, SLforge supports four more built-in libraries which our study identified as widely-used libraries. Additionally, using a customized Csmith (a powerful random C-program generator [27]), we introduced automated integration of custom block functionalities in the generated models, for the very first time.

Finding and Understanding Bugs. SLforge continuously generated models and tested Simulink for approximately five months. We have reported all except two of the 12 cases which SLforge identified as bugs; to date MathWorks has confirmed 10 of the reported issues as unique bugs, of which 8 were previously unknown.

We classified bugs based on the *essential* generator/differential testing feature that helped to discover them and observed that large, hierarchical model creation along with specification support and EMI-testing were the root causes of the bug discoveries [4]. The absence of these essential features limited CyFuzz's bug-finding power and consequently, the scheme was not successful in finding new bugs.

REFERENCES

[1] Angelo Brillout, Nannan He, Michele Mazzucchi, Daniel Kroening, Mitra Purandare, Philipp Rümmer, and Georg Weissenbacher. 2010. Mutation-Based Test Case Generation for Simulink Models. In *Formal Methods for Components and Objects: 8th International Symposium, FMCO 2009, Eindhoven, The Netherlands, November 4-6, 2009. Revised Selected Papers*, Frank S. de Boer, Marcello M. Bonsangue, Stefan Hallerstede, and Michael Leuschel (Eds.). Springer, 208–227.

[2] Shafiul Azam Chowdhury. 2018. Understanding and Improving Cyber-Physical System Models and Development Tools. In *ICSE '18 Companion: 40th International Conference on Software Engineering*. ACM.

[3] 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.

[4] 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 *40th International Conference on Software Engineering (ICSE 2018)*. ACM.

[5] Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: An automatic robustness tester for Java. *Software—Practice & Experience* 34, 11 (Sept. 2004), 1025–1050. https://doi.org/10.1002/spe.602

[6] Christian Dernehl, Norman Hansen, and Stefan Kowalewski. 2016. *Combining Abstract Interpretation with Symbolic Execution for a Static Value Range Analysis of Block Diagrams*. Springer International Publishing, Cham, 137–152. https://doi.org/10.1007/978-3-319-41591-8_10

[7] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust Typechecker Using CLP (T). In *Proc. 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 482–493. https://doi.org/10.1109/ASE.2015.65

[8] Frank Elberzhager, Alla Rosbach, and Thomas Bauer. 2013. 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 2013)*. ACM, New York, NY, USA, 29–34. https://doi.org/10.1145/2489280.2489285

[9] P. Fehér, T. Mészáros, L. Lengyel, and P. J. Mosterman. 2013. Data Type Propagation in Simulink Models with Graph Transformation. In *2013 3rd Eastern European Regional Conference on the Engineering of Computer Based Systems*. 127–137. https://doi.org/10.1109/ECBS-EERC.2013.24

[10] K. Ghani, J. A. Clark, and Y. Zhan. 2009. Comparing algorithms for search-based test data generation of Matlab (R) Simulink (R) models. In *2009 IEEE Congress on Evolutionary Computation*. 2940–2947. https://doi.org/10.1109/CEC.2009.4983313

[11] Antoine Girard, A. Agung Julius, and George J. Pappas. 2008. Approximate Simulation Relations for Hybrid Systems. *Discrete Event Dynamic Systems* 18, 2 (2008), 163–179. https://doi.org/10.1007/s10626-007-0029-9

[12] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proc. 21th USENIX Security Symposium*. USENIX Association, 445–458. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[13] Ishtiaque Hussain, Christoph Csallner, Mark Grechanik, Qing Xie, Sangmin Park, Kunal Taneja, and B.M. Mainul Hossain. 2016. RUGRAT: Evaluating program analysis and testing tools and compilers with large generated random benchmark applications. *Software—Practice & Experience* 46, 3 (March 2016), 405–431.

[14] 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.

[15] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core Compiler Fuzzing. In *Proc. 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 65–76.

[16] Reza Matinnejad, Shiva Nejati, Lionel C. Briand, and Thomas Bruckmann. 2016. SimCoTest: A test suite generation tool for Simulink/Stateflow controllers. In *Proc. 38th International Conference on Software Engineering, (ICSE)*. ACM, 585–588. https://doi.org/10.1145/2889160.2889162

[17] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107. http://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf

[18] Luan Viet Nguyen, Christian Schilling, Sergiy Bogomolov, and Taylor T. Johnson. 2015. HyRG: A random generation tool for affine hybrid automata. In *Proc. 18th International Conference on Hybrid Systems: Computation and Control (HSCC)*. ACM, 289–290. https://doi.org/10.1145/2728606.2728650

[19] Luan Viet Nguyen, Christian Schilling, Sergiy Bogomolov, and Taylor T. Johnson. 2015. Runtime Verification of Model-based Development Environments. In *Proc. 15th International Conference on Runtime Verification (RV)*.

[20] Vera Pantelic, Steven Postma, Mark Lawford, Monika Jaskolka, Bennett Mackenzie, Alexandre Korobkine, Marc Bender, Jeff Ong, Gordon Marks, and Alan Wassyng. 2017. Software engineering practices and Simulink: bridging the gap. *International Journal on Software Tools for Technology Transfer* (2017), 1–23. https://doi.org/10.1007/s10009-017-0450-9

[21] Jesse Ruderman. 2007. Introducing jsfunfuzz. https://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/. (2007).

[22] Prahladavaradan Sampath, A. C. Rajeev, S. Ramesh, and K. C. Shashidhar. 2007. Testing Model-Processing Tools for Embedded Systems. In *Proc. 13th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 203–214. https://doi.org/10.1109/RTAS.2007.39

[23] A. Sridhar, D. Srinivasulu, and D. P. Mohapatra. 2013. Model-based test-case generation for Simulink/Stateflow using dependency graph approach. In *Proc. 3rd IEEE International Advance Computing Conference (IACC)*. 1414–1419. https://doi.org/10.1109/IAdCC.2013.6514434

[24] Ingo Stürmer and Mirko Conrad. 2004. Code Generator Testing in Practice. In *INFORMATIK 2004 - Informatik verbindet (LNI)*, Vol. 51. GI, 33–37. http://subs.emis.de/LNI/Proceedings/Proceedings51/article3205.html

[25] Ingo Stürmer, Mirko Conrad, Heiko Dörr, and Peter Pepper. 2007. Systematic Testing of Model-Based Code Generators. *IEEE Transactions on Software Engineering (TSE)* 33, 9 (Sept. 2007), 622–634. https://doi.org/10.1109/TSE.2007.70708

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

[27] 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.