Finding Bugs in Cyber-Physical System Development Environments

Shafiul Azam Chowdhury[1], Siddhant Gawsane[1], and Sidharth Mehra[1]

The University of Texas at Arlington, Arlington TX 76019, USA,
`{shafiulazam.chowdhury,siddhant.gawsane,sidharth.mehra}@mavs.uta.edu`

Abstract. Over the years, cyber-physical system (CPS) development environments have facilitated the development of complex systems, by simulating their functionalities and automatically generating deployable artifacts. Bugs that might occur in a CPS development environment, can therefore cause expensive cost overheads at the very least and critical failures at the most. Differential testing has been proven to be of great merit in identifying bugs in these emulation environments. In our work we study Simulink, a highly popular tool used for modeling, simulating and analyzing control systems. Specifically we study defects in the Simulink subsystem to identify features that can be termed fault prone. We further extend the existing differential testing framework, CyFuzz, to target these features and using random input generation to find an exhaustive and comprehensive list of possible bugs in Simulink.

Keywords: Differential testing, cyber-physical systems, Simulink

1 Introduction

CPS development environments are used to design intricate and large-scale control systems that are often used in mission-critical and safety-critical environments. Previous defect analysis studies on CPS Development Environments show that defects in the environment can lead to defects in the system design, which show up as erroneous behavior in the control system. Defects in a hardware system can lead to massive cost overheads, product recalls and safety and compliance issues. Fixing bugs in these CPS Design Environments can therefore prove to be of drastic significance.

A CPS Design Environment, such as Simulink, is used to design, emulate and tune the CPS design. It follows a model based architecture where blocks, models and subsystems are used to comprehensively describe the target system. Various simulators are available that can be used to simulate the working of the system as is would in the real world. Finally various verification and validation tools can be used, such as model checking, automated test case generation, hardware-in-the-loop and software-in-the-loop testing etc., to make sure that only valid models are generated and generic errors in design are pointed out. It might seem surprising that despite of having such advanced simulation methods, products often fail in real-time and key design defects emerge only once the system is live.

Various studies show that some of these defects can be back traced to bugs in the CPS Development System itself.

A technique called formal verification is a popular methodology which entails rigorously monitoring each input path in each sequence of blocks for every possible model. There are, however, multiple hurdles with this approach. Firstly, detailed specifications are not available for each block type, possibly because of functional complexity for each type of block, or due to fast release cycles. Secondly, there is no upper limit on the number of types of blocks that can be used to design a model. And lastly, the possible permutations and combinations of block connections can grow infinitely large, and it would be impossible to estimate the computing power required to simulate a model of a random size and random specification. Instead, we choose an alternative less rigorous approach called differential testing or fuzzing, which is a form of random testing, that essentially attempts to execute models with a random sequence of inputs.

This technique is been proven to show promising results with compilers of traditional programming languages, like Csmith [1], jsfunfuzz [2], and LangFuzz [3] that have collectively found over 1,000 bugs, even in widely used compilation tools, such as GCC [4]. It has also shown considerable results in our previous work, CyFuzz, which has been shown to demonstrate impressive results within various features in Simulink [5].

Our present focus is to expand on CyFuzz's success by analyzing defect reports in Simulink to pick out areas that can be claimed to be defect prone. We extend CyFuzz's functionality to address these target areas, hence enabling better coverage over possible input values. Among these, a particular type of target environment called the Embedded Coder was special case in study. Our work includes successfully extending CyFuzz to target this environment. We've also addressed pre-existing limitations with hierarchical models that greatly inhibited CyFuzz's ability to generate complex models that are comparable with real-world models. Also noteworthy in our contribution is a metric based analysis of various models, available as part of the Simulink package, open source models obtained from public forums like github and models generated previously by CyFuzz itself.

Our work in the expansion of CyFuzz has proven effective in improving it's capabilities. Of all the bugs reported, 4 have been validated and accepted as unique, previously unknown bug by MathWorks technical support team.

2 Related Works

Differential testing is common technique that finds applications in compiler fuzzers. Broadly classified, these can be of two types. Compilers for common programming languages can be verified using semantic fuzzers. The most well known is CSmith[1] which verifies any given C compiler against the C99 standard. Data-flow languages, however, are tested using data-flow fuzzing, as used by CyFuzz[5]. Input generation and comparison for both these work significantly differently.

Our work heavily draws on Vu Le et al. and their analysis of Equi-Modulo Inputs to verify different compilers[6]. Also related is Nyugen et al.[7] and Csallner et al.[8] and their work on a differential testing based runtime verification framework, as well as Sampath et al.'s approach in testing model-processing tools by using Stateflow(a Simulink component)'s semantic meta-model[9].

References

1. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), ACM (June 2011) 283–294
2. Ruderman, J.: Introducing jsfunfuzz. `https://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/` (2007)
3. Holler, C., Herzig, K., Zeller, A.: Fuzzing with code fragments. In: Proc. 21th USENIX Security Symposium, USENIX Association (August 2012) 445–458
4. Dewey, K., Roesch, J., Hardekopf, B.: Fuzzing the Rust typechecker using CLP (T). In: Proc. 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE (2015) 482–493
5. Chowdhury, S.A., Johnson, T.T., Csallner, C.: 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), Cham, Springer International Publishing (October 2016) 46–60
6. Le, V., Afshari, M., Su, Z.: Compiler validation via equivalence modulo inputs, ACM (April 2014) 289–290
7. Nguyen, L.V., Schilling, C., Bogomolov, S., Johnson, T.T.: HyRG: A random generation tool for affine hybrid automata. In: Proc. 18th International Conference on Hybrid Systems: Computation and Control (HSCC), ACM (April 2015) 289–290
8. Csallner, C., Smaragdakis, Y.: JCrasher: An automatic robustness tester for Java. Software—Practice & Experience **34**(11) (September 2004) 1025–1050
9. Sampath, P., Rajeev, A.C., Ramesh, S., Shashidhar, K.C.: Testing model-processing tools for embedded systems. In: Proc. 13th IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE (April 2007) 203–214