Automatic Generation of Simulink Models to Find Bugs in a Cyber-Physical System Tool Chain using Deep Learning

Sohil Lal Shrestha
University of Texas at Arlington
Arlington, TX, USA

ABSTRACT

Testing cyber-physical system (CPS) development tools such as MathWorks' Simulink is very important as they are widely used in design, simulation, and verification of CPS data-flow models. Existing randomized differential testing frameworks such as SLforge leverages semi-formal Simulink specifications to guide random model generation which requires significant research and engineering investment along with the need to manually update the tool, whenever MathWorks updates model validity rules. To address the limitations, we propose to learn validity rules automatically by learning a language model using our framework DeepFuzzSL from existing corpus of Simulink models. In our experiments, DeepFuzzSL consistently generate over 90% valid Simulink models and also found 2 confirmed bugs by MathWorks Support.

ACM Reference Format:
Sohil Lal Shrestha. 2020. Automatic Generation of Simulink Models to Find Bugs in a Cyber-Physical System Tool Chain using Deep Learning. In *42nd International Conference on Software Engineering Companion (ICSE '20 Companion), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3377812.3382163

1 INTRODUCTION AND MOTIVATION

Engineers often prototype cyber-physical systems (CPS) such as fuel management system in airplanes using commercial tool chains such as MathWorks Simulink [12], which enable them to model, simulate and analyze their system using graphical block diagrams. Furthermore, these tool can automatically generate executables that can be directly deployed to target hardware. Since these executable are often deployed in safety critical systems, it is of immense importance that the tool chains are bug-free, as a bug in the tool chain can propagate to the artifacts they produce.

Randomized differential testing is an effective way to test complex software system that eliminates the need of complete specification as well as test oracle. Earlier work by Chowdhury et.al has been able to find 8 new bugs in Simulink using their tool *SLforge* [4] and 9 confirmed bugs using SLEMI [6] which uses *SLforge*'s generated Simulink models as seed.

Chowdhury et al. gather and interpret semi-formal specifications from Simulink's web page and incorporated them in *SLforge*'s random model generator. While this approach has been proven effective, *SLforge*'s random model generator requires significant research and engineering investment along with the need to manually update the tool whenever Simulink adds or changes its model validity rules silently. And the engineering efforts do not translate well to other tool chains, as each has its own model validity rules. Furthermore, it is also desired that randomly generated models be similar to ones created by humans as bugs in such models are likely to be fixed by the system under test (SUT) developers.

To side step the limitation of random model generation, we propose to automatically infer validity rules from existing Simulink models by training a language model of Simulink models using deep learning techniques (aka neural language models). This approach will make re-learning rules easier even if the underlying language rules change or transfer the same approach to a different language. Similar approach has already been applied for compiler fuzzing of imperative language. Their framework DeepSmith [8] learned from over 10,000 real OpenCL programs reporting over 50+ bugs in OpenCL compilers and claimed that they are easily extensible to other programming languages with minimum engineering efforts.

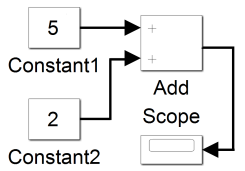

Figure 1: Simulink model encoded in Listing 1.

```
Model { ...
Block {
  BlockType  Constant
  Name  "Constant1"... }
Block {
  BlockType  Constant
  Name  "Constant2"... }
Line {
  SrcBlock "Constant1"
  DstBlock "Add" ... }
```

Listing 1: Simulink model encoding of Figure 1

Our work is motivated by the language agnostic framework of DeepSmith and we extend their work for Simulink models. Unlike imperative language which are closely related to natural language both in terms of program statements as well as having rigid syntactic and semantics structure, CPS developers design Simulink models using graphical block diagrams as seen in Figure 1. Equivalent textual representation can easily extend over 1,000 LOC with over 1000 keywords even for simple toy example which does not translate well with learning language model using current deep learning techniques. We tackle the issues with DeepFuzzSL framework. In our experiment, our trained DeepFuzzSL model is able to generate over 90% valid Simulink models and have found 2 confirmed bugs in Simulink.

2 BACKGROUND

Language modeling is a task of predicting the next word given a set of previous words. More formally, given a sequence of words

ICSE '20 Companion, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7122-3/20/05…$15.00
https://doi.org/10.1145/3377812.3382163

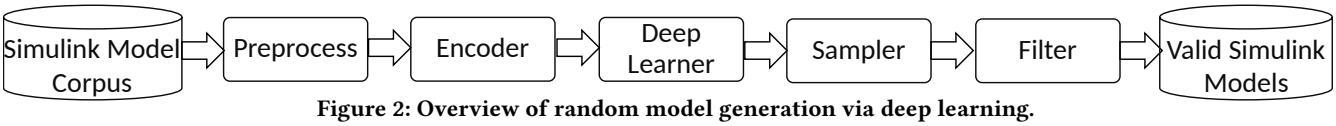

Figure 2: Overview of random model generation via deep learning.

TSC	Summary	MW	Kind
03632450	Simulink's parser fails to reject ill-formed model and crashes	K	O
03322011	Simulink's parser fails to reject model with ill-configured Signal Generator Block	N	S

Table 1: Summary of bugs, *TSC* = *Technical Support Case* number from MathWorks; *MW* = feedback from MathWorks on bug report (*N* = Likely new bug, *K* = known bug); *O* = bug reported when opening model; *S* = bug reported when simulating model;

$x^1, x^2 \ldots x^t$, a language model computes probability distribution of next word x^{t+1} as $P(x^{t+1}|x^t \ldots x^1)$, where x^{t+1} can be any word in a vocabulary V = $\{w_1, \ldots w_{|V|}\}$. Neural language model refers to using neural network architecture to learn language models [2].

Program statements of languages like C are closely related to natural language, which has rigid syntax and semantic. Thus, language models can be directly learned from a corpus of handwritten programs. Although a CPS is designed as a set of data flow graphical *models* as seen in Figure 1, their textual representation does follow the norm and can be used as a training corpus. A typical CPS model contains *blocks* that accepts data through the *input* ports. Block performs some operation on them and pass output to other blocks through *output* ports, using *connection* lines. Simulink is a de-facto standard commercial tool chain for CPS supporting various built-in block *libraries* [17]. A model is said to be valid if it can be compiled by Simulink without errors. Formal and in depth description of CPS and Simulink can be found elsewhere [5, 13].

3 OVERVIEW AND DESIGN

Figure 2 represent phases of learning a generative model and sampling from the trained model to generate valid Simulink models.

Simulink Model Corpus: Simulink model files are typically distributed in new standard slx format, that has separate xml files describing the Simulink model component, or mdl format which was MathWork's proprietary file format before 2012. The main difference being mdl file consist of all the necessary model parameter definition in a single text file while slx has a definition maintained in different file. We restricted ourselves in studying and using only mdl files in our training corpus since, self-sufficient nature of mdl makes it easy to generate Simulink model and there is support for converting mdl to slx format if required.

Pre-processing: Before feeding the seed corpus of Simulink models to the neural network for training, we perform pre-processing steps so that we don't overburden the neural network to learn unnecessary rules that do not contribute to generate a valid Simulink model. We carry basic pre-processing steps such as white space removal, converting long block name to shorter ones, removing any annotations and location information as they are not required for its validity. Since neural network learns sequential information, we also interleave block and connection (or line) information in the text file of Listing 1.

Encoder: Neural network requires numeric sequences as inputs. Hence all Simulink model (mdl) file is converted to a sequence of fixed length feature vectors. To limit the size of resulting feature vector, we encode Simulink models using a hybrid scheme that maps a few common keywords to tokens and the rest to characters.

Neural Network: We use Long Short Term Memory networks, variant of recurrent neural networks following success of many recent work [8, 11, 14]. We use two layer LSTM network with 512 nodes per layer which strike a balance between size of the neural net and the closeness of the learned distribution to the true distribution resulting in practical training time of neural network. We defined the model using Keras [3] and Tensorflow [1] and open sourced it on Github [15] for other researcher to train their own corpus.

Sampler: After the training is complete, we seed the trained neural net with "Model {" keyword and then sample token-by-token to generate Simulink models. Since we want to maximize the number of variations of generated models, we chose the next token from randomized learned distribution by performing a multi-nominal experiment. Generation of model file is complete when the opening and closing bracket count become balanced or it reaches maximum number of allowed tokens.

Filter: Lastly we filter out the generated Simulink models by opening, compiling and simulating them in Simulink.

4 EXPERIMENTAL SETUP AND EVALUATION

Since deep learning models require large seed corpus and the publicly available Simulink's model [7] are limited, we trained our LSTM network using 1,000 *SLforge*'s generated seed corpus on Texas Advanced Computing Center (TACC) [16], which provides high performance computing resources. We trained the network for 400 epoch using gradient descent with a learning rate of 0.002, decaying 5% every epoch and Adam optimizer [10] with 64 mini-batch size. We selected the hyper-parameters based on the best result after multiple runs of the experiment. On TACC's maverick2 GTX nodes, training the neural network took about 2 hours.

To evaluate our approach, we sampled 1,024 Simulink models from trained LSTM network limiting maximum size of tokens in each generated sample at 5,000 (corresponding to seed corpus's size) and reported ratio of valid Simulink models (i.e., models that compiles) using different sampling strategies described in [9]. The sampling time for each sampling strategies took around 13 minutes. We observed over 90% valid generated Simulink model by our approach. Furthermore, we also reported 2 unique issues to *MathWorks support*, both of which they confirmed as bug in Simulink tool-chain summarized in Table 1. All bugs exist in Simulink R2019a.

REFERENCES

[1] Martin Abadi et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). http://tensorflow.org/ Software available from tensorflow.org.

[2] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. 2001. A Neural Probabilistic Language Model. In *Advances in Neural Information Processing Systems 13*, T. K. Leen, T. G. Dietterich, and V. Tresp (Eds.). MIT Press, 932–938. http://papers.nips.cc/paper/1839-a-neural-probabilistic-language-model.pdf

[3] François Chollet et al. 2015. Keras. https://keras.io. (2015).

[4] Shafiul Azam Chowdhury. 2018. Automatically Finding Bugs in Commercial Cyber-physical System Development Tool Chains. In *Proc. 40th International Conference on Software Engineering (ICSE): Companion Proceeedings*. ACM, 506–508.

[5] 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 ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, 981–992.

[6] Shafiul Azam Chowdhury, Sohil L Shrestha, Taylor T. Johnson, and Christoph Csallner. To appear. SLEMI: Equivalence Modulo Input (EMI) Based Mutation of CPS Models for Finding Compiler Bugs in Simulink. In *Proc. 41st ACM/IEEE International Conference on Software Engineering (ICSE)*.

[7] Shafiul Azam Chowdhury, Lina Sera Varghese, Soumik Mohian, Taylor T. Johnson, and Christoph Csallner. 2018. A curated corpus of Simulink models for model-based empirical studies. In *Proc. 4th International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS)*. ACM, 45–48.

[8] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler fuzzing through deep learning. In *ISSTA 2018*.

[9] Ari Holtzman, Jan Buys, Maxwell Forbes, and Yejin Choi. 2019. The Curious Case of Neural Text Degeneration. *CoRR* abs/1904.09751 (2019). arXiv:1904.09751 http://arxiv.org/abs/1904.09751

[10] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. (2014). http://arxiv.org/abs/1412.6980 cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[11] Xuan Liu, Di Cao, and Kai Yu. 2018. Binarized LSTM Language Model. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. Association for Computational Linguistics, New Orleans, Louisiana, 2113–2121. https://doi.org/10.18653/v1/N18-1192

[12] MathWorks Inc. 2019. MATLAB & Simulink. https://www.mathworks.com/products/simulink.html/. (2019). Accessed Jan 2020.

[13] MathWorks Inc. 2019. Simulink Documentation. https://www.mathworks.com/help/simulink/. (2019). Accessed Jan 2020.

[14] Alec Radford, Rafal Józefowicz, and Ilya Sutskever. 2017. Learning to Generate Reviews and Discovering Sentiment. *CoRR* abs/1704.01444 (2017). arXiv:1704.01444 http://arxiv.org/abs/1704.01444

[15] Sohil L Shrestha. 2019. DeepFuzzSL. https://github.com/50417/DeepFuzzSL/releases. (2019). Accessed Jan 2020.

[16] TACC at The University of Texas at Austin. 2018. Texas Advanced Computing Center - Homepage. https://www.tacc.utexas.edu/. (2018). Accessed Jan 2020.

[17] M. Wolf and E. Feron. 2015. What don't we know about CPS architectures?. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–4. https://doi.org/10.1145/2744769.2747950