SLGPT: Using Transfer Learning to Directly Generate Simulink Model Files and Find Bugs in the Simulink Toolchain

Sohil Lal Shrestha
Computer Science and Engineering Department
University of Texas at Arlington
Arlington, Texas, USA

Christoph Csallner
Computer Science and Engineering Department
University of Texas at Arlington
Arlington, Texas, USA

ABSTRACT

Finding bugs in a commercial cyber-physical system (CPS) development tool such as Simulink is hard as its codebase contains millions of lines of code and complete formal language specifications are not available. While deep learning techniques promise to learn such language specifications from sample models, deep learning needs a large number of training data to work well. SLGPT addresses this problem by using transfer learning to leverage the powerful Generative Pre-trained Transformer 2 (GPT-2) model, which has been pre-trained on a large set of training data. SLGPT adapts GPT-2 to Simulink with both randomly generated models and models mined from open-source repositories. SLGPT produced Simulink models that are both more similar to open-source models than its closest competitor, DeepFuzzSL, and found a super-set of the Simulink development toolchain bugs found by DeepFuzzSL.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; *Model-driven software engineering*; • **Computing methodologies → Transfer learning**; • **Information systems → Language models**.

KEYWORDS

Cyber-physical system development, Simulink, tool chain bugs, deep learning, programming language modeling, GPT-2

ACM Reference Format:
Sohil Lal Shrestha and Christoph Csallner. 2021. SLGPT: Using Transfer Learning to Directly Generate Simulink Model Files and Find Bugs in the Simulink Toolchain. In *Evaluation and Assessment in Software Engineering (EASE 2021), June 21–23, 2021, Trondheim, Norway*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3463274.3463806

1 INTRODUCTION

Finding bugs in a commercial cyber-physical system (CPS) development tool such as Simulink is hard as its codebase contains millions of lines of code and complete formal language specifications are not available. While deep learning techniques promise to learn such language specifications from sample models, deep learning needs a large number of training data to work well and the closest related deep learning tool DeepFuzzSL [21] is severely limited by the relatively small number of available training models.

Testing CPS development tools is important as engineers design and develop dynamic safety-critical systems using these development tools. For example, MathWorks's Simulink is widely used in industry such as automotive, medical, and aerospace [25]. Engineers use Simulink to design, simulate, test, and generate embedded code from CPS models and deploy it to end-user hardware. At worst a subtle bug in the Simulink tool chain could result in unexpected behaviour in safety-critical applications such as in cars or airplanes.

Given the complexity of the Simulink language, training a deep learning tool such as DeepFuzzSL from scratch would require a very large number of training models. However relatively few open source Simulink models are available. While random model generators such as SLforge [2] could fill in some of these gaps, it is not clear how well SLforge can cover the various features (and their combinations) of the Simulink language.

Given the limited amount of Simulink training models, this paper proposes to use transfer learning for generating Simulink models. Transfer learning is a promising alternative to learning from scratch, as it leverages a machine learning model trained on a large set of related training data. We can then use a relatively small set of Simulink-specific training data to fine-tune such a pre-trained model for generating Simulink models.

Precisely we fine-tune the Generative Pre-trained Transformer 2 (GPT-2) [17] model using both randomly generated models and models we mined from the open-source repositories GitHub and MATLAB Central. Our experimental results suggest that GPT-2 generated Simulink models are of higher quality and address the shortcomings of earlier deep learning approaches. SLGPT also found a wider range of similar bugs found by DeepFuzzSL in Simulink versions R2018b, R2019b, and R2020b confirmed by Mathworks Support. To summarize, the paper makes the following contributions.

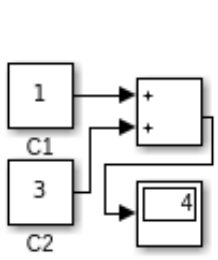
- SLGPT is the first use of transfer learning for generating graphical block-diagram models.
- The paper implements SLGPT for Simulink, collects a training set of 400 open-source Simulink models, and compares SLGPT with the closest related tool DeepFuzzSL.
- SLGPT-created models were more similar to open-source models and SLGPT found a super-set of the Simulink development toolchain bugs DeepFuzzSL found.
- The SLGPT implementation, parameter settings, and training sets are open-source [20].

2 BACKGROUND

Simulink [15] is a powerful commercial tool-chain for model-based design and has become a de-facto standard in several domains

EASE 2021, June 21–23, 2021, Trondheim, Norway

ACM ISBN 978-1-4503-9053-8/21/06.
https://doi.org/10.1145/3463274.3463806




```
1   Model { Name "toy"   ...
2     System { ...
3       Block {
4         BlockType     Sum
5         Name   "Add"  ... }
6       Block {
7         BlockType Constant
8         Name    "C1"  ... }
                ...
9       Line {
10        SrcBlock     "C1"
11        DstBlock     "Add"
12        ... } ...   } }
```

Figure 1: Simulink model (left) and excerpt of its 1.1k line (Simulink-generated) model file representation (right).

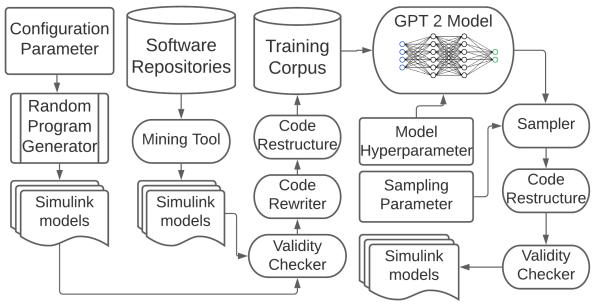


Figure 2: SLGPT obtains Simulink models from a random generator and open-source repositories, simplifies them, and uses them to adapt GPT-2 for finding Simulink crashes.

such as automotive and aerospace. An engineer typically designs a model via Simulink's graphical modeling environment. A Simulink model is a (potentially hierarchical) *block diagram*, where each block represents equations or modeling components. A Simulink user can also define *custom blocks* in custom "native" code using the S-function interface. Simulink typically stores a model in its proprietary model file format, i.e., as a structured ASCII file that contains keywords and parameter-value pairs (many of which are case-sensitive) [12]. Figure 1 shows a flat Simulink model and parts of its model file representation.

Depending on the block type, each block can accept input via input ports, perform some operation on its inputs, and pass output via output ports to other blocks through (directed) edges. Simulink users can add blocks from various built-in *libraries* and toolboxes. A source block generates signals in a Simulink model while a sink block is used to display or output signals[14]. A model's maximum source-to-sink path length is the longest directed non-circular path from a source to a sink node (and includes source and sink).

When a user opens a model, Simulink's parser performs its checks and prevents corrupt models from opening. Once opened, a user can compile and then simulate the model, where the tool chain uses configurable solvers to iteratively solve the model's network of mathematical relations via numerical methods, yielding for each output block a sequence of outputs. After simulation, the user may use Simulink's embedded code generation workflow for deployment on a target platform.

2.1 Transfer Learning & NLP Language Models

Transfer learning [16] is a promising technique for generating Simulink models, as transfer learning can work well in scenarios that suffer from relatively small amounts of training data. Transfer learning achieves this by using a machine learning model trained for a source task or domain ("pre-training", e.g., programs in any programming language) as a starting point to train on a different but related target task or domain ("fine-tuning", i.e., Simulink models). This works well if pre-training uses huge amounts of training samples, learns features common to both tasks, and fine-tuning can apply the learned knowledge on a target task. Successful applications include computer vision, where large datasets such as ImageNet [6] have been used to pre-train deep learning models that are later fine-tuned for tasks such as image segmentation.

In natural language processing (NLP), language modeling is the use of statistical techniques to determine the probability of a given word sequence. A language model basically estimates the probability of a word based on the words already observed in a sequence. An effective language model not only understands language structure (syntax) but also long-term context (semantics). For example, a Simulink language model should predict tokens that are both syntactically correct and produce valid connections between blocks (e.g., respecting Simulink language rules on define-before-use).

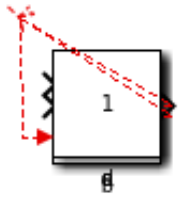
Transfer learning in natural language processing is relatively new. ULMFiT presents a specific training schedule enabling transfer learning using LSTMs [10]. GPT-2 uses transformer decoder as a building block and trains a language model on the WebText dataset [17]. Using transformers instead of LSTMs allows longer-range context capture. GPT-2's byte pair encoded vocabulary also supports Unicode (and does not require common pre-processing steps such as lower-casing and stemming). So GPT-2 is a great candidate to learn Simulink model files.

3 OVERVIEW AND DESIGN

Figure 2 gives an overview of SLGPT. To obtain a variety of Simulink models for machine learning, we both ran the random model generator SLforge and mined open-source repository sites, i.e., GitHub and MATLAB Central. Since GitHub currently does not treat Simulink as a searchable language, we used the GitHub API with "Simulink" as search keyword. Since MATLAB Central does not provide an API for downloading Simulink models, we used its RSS feed[1] to heuristically construct Simulink project download links.

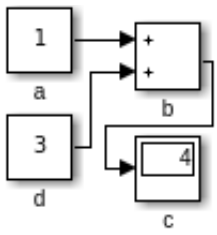
We want our training corpus to only contain valid Simulink models. So we automate the process of checking if a Simulink model is compilable on Simulink. The validity checker also helps detect any crashes caused by an input Simulink model, which is then manually reviewed and reported to the developers. To limit the number of Simulink language features in our training data, we only used flat models that do not have additional toolbox or library dependencies, yielding 400 valid open-source Simulink models for training.

[1] https://www.mathworks.com/company/rss.html




```
1   Model { Name "toy"
2   System {
3   Block {
4   BlockType Constant
5   Name "a" ... }
6   Line {
7   SrcBlock "a"
8   DstBlock "b"
9   ... }
10  Block {
11  BlockType Sum
12  Name "b"
13  ... }...}}
```

Figure 3: Figure 1 Simulink model and excerpt of its model file, after SLGPT simplified it to 45 lines, by removing layout info, restructuring code via BFS, etc.




```
1   Model { Name "toy"
2   System {
3   Block {
4   BlockType Constant
5   Name "a" ... }
6   Block {
7   BlockType Sum
8   Name "b" ... }...
9   Line {
10  SrcBlock "a"
11  DstBlock "b"
12  ... }...}}
```

Figure 4: Figure 3 Simulink model and excerpt of its 45 line model file, after SLGPT restored it to Simulink-compliant style (plus manual layout changes for readability).

3.1 Training Data Preparation: Simplification

SLGPT simplifies training models to (1) remove model features we currently cannot handle given the limited number of training models and to (2) restructure models to fit GPT-2's learning style. While both simplification types may change model semantics, SLGPT compensates for type-2 simplifications (restructuring), by rewriting generated models into equivalent Simulink-compliant style.

Specifically, we pre-process the model file to remove macros, default configuration settings, comments, duplicate white spaces, annotations, and block-position information. We similarly rewrite model identifiers (e.g., block names) to short but unique names (a, b, c, …, aa, ab, ac, …), based on their appearance order in our restructured model file.

The ASCII style in which Simulink saves its models to files is problematic for state-of-the-art deep learning language models, as Simulink files are long and verbose. Furthermore, these files also list all nodes before all edges. Taken together, this is a poor fit for current language models, which model context with a text window of limited size.

To make such files easier to learn, SLGPT's Algorithm-1 rewrites these files in a breath-first search (BFS) style. Specifically, we first parse the Simulink model file and maintain an adjacency representation of the Simulink model in the $graph\_info$ map, which maintains two disjoint sets: $source\_blks$ has blocks with $in\_degree = 0$ and other $other\_blks$ has all remaining blocks. Algorithm-1's outer loop iterates over both S and B as some Simulink models have dangling blocks (blocks that are not source blocks and are not connected to any other block). Also some models (especially from SLforge) have no source blocks because they have cycles.

Algorithm 1: Restructuring Simulink model. "Neighbour" refers to both incoming and outgoing blocks and edges.

Require: $source\_blks$ (S), $other\_blks$ (B), $graph\_info$ (G)
Result: BFS-rewrite of Simulink model file (C_{BFS})

```
1  while S ≠ ∅ and B ≠ ∅ do
2      Q = empty queue
3      b = remove element from (S ≠ ∅)?S : B
4      add b to back of Q
5      while Q ≠ ∅ do
6          curb = pop element from front of Q
7          if curb ∉ C_BFS then
8              add curb to C_BFS
9              remove curb from B
10             B_nei, E_nei = curb's neighbour blocks, edges in G
11             forall e ∈ E_nei do
12                 if e ∉ C_BFS then
13                     add e to C_BFS
14                 end
15             end
16             forall b ∈ B_nei do
17                 if b ∉ C_BFS then
18                     add b to back of Q
19                 end
20             end
21         end
22     end
23 end
```

3.2 Synthesizing Simulink Models with GPT-2

Given the complexity of the Simulink language, generating valid Simulink model files is an ambitious task for unsupervised machine learning, especially given our small amounts of training data. Instead of training from scratch, we thus use the pre-trained language model GPT-2. GPT-2 is a good fit, as it employs byte pair encoding to construct its vocabulary, meaning all tokens in a Simulink model file can be mapped to the vocabulary set.

Second, GPT-2's architecture is based on the transformer architecture [23], which has benefits over a traditional LSTM architecture, as transformers avoid recursive computation by processing sequences as a whole and learning relationships between tokens by using multi-head attention mechanisms and positional embeddings. This enables better prediction, which is typically lost with LSTM over long-term dependencies in the text.

SLGPT's Algorithm-2 iteratively samples from the fine-tuned language model to generate Simulink model files. We seed the model with the sequence "Model {" and then sample token by token. In this early project stage we followed the best sampling techniques of DeepFuzzSL (nucleus or top-p sampling [9]). Specifically, given a

start text S, sampling parameters nucleus N and temperature T, the fine-tuned GPT-2 model G computes the probability mass function PMF representing the probability distribution of all tokens in the vocabulary. We normalize the PMF after scaling with T to introduce randomness. To reduce the size of next plausible tokens, we select the smallest subset of PMF such that the sum of all values in the subset is greater than N. The normalized subset PMF is then used to perform a multinomial experiment to choose the next token.

Algorithm 2: Sampling a candidate Simulink model from a seed text.

Require: Fine-tuned GPT-2 model (G), temperature (T), nucleus (N)
Result: Completed sample string S

```
S = "Model { "
while <endoftext> ∉ S do
    PMF = get_distribution_of_next_predicted_tokens(G, S)
    Scale the obtained PMF by T
    Sort PMF in descending order
    Subset PMF such that the smallest possible set sum is
      greater than N
    R = Perform multinomial experiment on subset PMF
    S = S + R
end
```

Since the resulting Simulink model file S is (as the training samples) in BFS style (as Simulink expects block definitions before edge definitions in a model file), SLGPT restructures S such that the model defines all blocks before defining edges. To continue the Figure 1 example, if we assume Figure 3 shows a model produced by Algorithm-2, SLGPT then reorders its element definitions to the Simulink-friendly style of Figure 4.

In lieu of full differential testing, SLGPT just uses its validity checker to detect crashes of the Simulink tool. We then manually investigate each crash, judge if a crash is an example of a known bug, and report representatives of the remaining crashes to MathWorks Customer Support.

4 INITIAL EXPERIENCE

While a full evaluation is future work, this paper compares SLGPT to its most closely related competitor, i.e., DeepFuzzSL. We first used DeepFuzzSL's evaluation setup of a SLforge-generated training corpus, in which each Simulink model has 5–57 blocks. SLGPT's pre-processing reduced the number of tokens by 75%, yielding 987 Simulink models with a total of 0.5M lines. We ran a related experiment on the 400 open-source Simulink models. SLGPT's pre-processing removed the 23 of the 400 models that only contained annotation blocks, yielding 3.5k blocks represented in 87k lines.

OpenAI has released four different sizes of pre-trained GPT-2 models ranging from 0.1 to 1.5 billion parameters. To limit computational resource needs, for these initial experiments we used the smallest model. We fine-tuned the GPT-2 model remotely in the high performance Texas Advanced Computing Center (TACC)'s Maverick 2 cluster [22]. We ran our experiments on a single Maverick 2 GTX node[2] of two 8-core 2.1 Ghz Intel Xeon processors, 128 GB RAM, and 4 NVidia 1080-TI GPUs.

As in DeepFuzzSL's experimental setup we used the Adam optimizer (here to fine-tune the GPT-2 model). We could not use a mini batch size of 64 as it triggered out-of-memory errors on TACC. Instead, we used batch size of 1. To compensate for the low batch size, we set the learning rate to 0.00002 (vs. 0.002) and trained SLGPT for 24 hours on SLforge-generated models and in a separate experiment for 24 hours on the open-source models.

We trained DeepFuzzSL on the same hardware as SLGPT but otherwise as described in its paper, i.e., for 400 epochs with mini batch size 64. While this "only" took about 6.5 hours, DeepFuzzSL's loss function tapered off after 100–150 epochs (so it was not learning much after that). While sampling, we let DeepFuzzSL run until it either emits a terminating token or reached 15k tokens (corresponding to the largest open-source training model). In the latter case the resulting file typically contained several model-start sequences. When opening such a file, Simulink and our counts just ignore all but the first model. We use the following research questions.

RQ1 Can SLGPT generate valid Simulink models? How does the structure of DeepFuzzSL and SLGPT generated Simulink models compare to open-source models?

RQ2 How do DeepFuzzSL and SLGPT compare in the bugs they find in the Simulink tool chain?

4.1 SLGPT Can Generate Valid and More Realistic Simulink Models (RQ1)

Earlier approaches were evaluated in terms of the validity of generated models and their bug-finding ability (e.g., in SLForge, SLEMI, and DeepFuzzSL [2, 3, 21]). In addition, to evaluate the quality of a model generator, we compare structural properties of the generated Simulink models against open-source Simulink models. Specifically, we use the number of nodes in the generated Simulink model and metrics based on the common notion of a connected subgraph (i.e., a subgraph in which each node is connected to at least one other node in the subgraph).

To explore SLGPT's ability to generate valid Simulink models, we continuously generate Simulink models for 24 hours. Sampling the version trained on SLforge-generated models yielded 2,912 Simulink models of which 43% compiled. The version trained on open-source models yielded 709 Simulink models of which 47% compiled. The most frequent cause of compile errors include data type mismatches between two connecting blocks and assigning an alphanumeric value to a numeric block attribute. Most of these could be fixed easily by adding data type conversion blocks to the model and changing alphanumeric to numeric values.

We trained DeepFuzzSL on the same training sets as SLGPT and sampled around 1k samples each with DeepFuzzSL's sampling heuristics. To make the comparison consistent we removed DeepFuzzSL's output token bound and allowed DeepFuzzSL to generate complete Simulink model files. Of around 1,200 DeepFuzzSL-generated models trained on Slforge-generated models, 89% compiled, closely aligning with the 90% validity rate reported in the

[2] https://portal.tacc.utexas.edu/user-guides/maverick2

DeepFuzzSL paper. On the other hand, out of 1,024 DeepFuzzSL-generated Simulink models trained on open-source models only 42% compiled.

The valid models generated by DeepFuzzSL were not as similar to the training models as SLGPT-generated valid models. Figure 6 compares these models along four metrics. For example, DeepFuzzSL-generated models tend to have many subgraphs that only contain 2 blocks, many blocks have unconnected input and output ports, and there is often no connection between source and sink.

4.2 SLGPT Found Superset of Bugs DeepFuzzSL Found (RQ2)

Trained on SLforge-generated Simulink models, from nearly 3k SLGPT-generated models 13 crashed Simulink. Upon analysis these 13 instances belong to the same two bug categories DeepFuzzSL found (MathWorks confirmed both types as known bugs). The first issue is a Simulink crash while opening a model. The second issue is Simulink opening a model but crashing while compiling the model.

Trained on our open-source models, 30 DeepFuzzSL-generated and 14 SLGPT-generated models crashed Simulink while compiling. 13 of the SLGPT-generated (and all DeepFuzzSL-generated) models get rejected by Simulink R2018b but crash version R2020b (case 04777147). The last one crashes R2018b but is accepted by R2020b (case 04767975). Following are brief summaries of these two cases.

4.2.1 Case 04777147 (Non-bug). This SLGPT-generated Simulink model triggered an interesting behavior, where Simulink R2018b rejects it as corrupt and the newer R2020b version crashes. MathWorks told us that the way Simulink parses MDL files has changed since R2020a, which may have caused the crash. As it is impossible to create this model via Simulink's graphical editor or standard API, MathWorks Support marked it as a non-bug. DeepFuzzSL-generated models triggered similar Simulink crashes.

Figure 5: Scope (left) and Floating Scope (right).

4.2.2 Case 04767975 (Known bug). Figure 5 shows two types of Simulink scope blocks: Scope and Floating Scope. Floating Scope does not have any physical ports while Scope does. A SLGPT-generated model set floating parameter off (indicating that it is a normal scope) while setting the ports attribute to 0 (instead of a vector), causing the crash. Simulink's graphical editor or standard API cannot create this model. This issue exists in R2018b and has been fixed in later versions. DeepFuzzSL did not trigger this bug.

5 RELATED WORK

Small training datasets are a common problem in deep learning applications. Researchers thus often use synthetic datasets [7]. Robbes et. al. showed a promising avenue to alleviate the dataset problem by using transfer learning [19], i.e., that a small natural-language software engineering dataset can be used to improve sentiment analysis using pre-trained neural networks.

In the CPS domain, Chowdhury et al. developed a randomized differential testing tool using semi-formal specifications to test the Simulink toolchain [2]. Subsequently SLEMI generated semantic-preserving mutants of a seed model for differential testing of the Simulink toolchain [3]. While these approaches are tightly coupled with Simulink, SLGPT is only loosely coupled and does not rely on explicit Simulink language specifications.

Success of modeling natural language using deep learning has garnered interest to model source code for program generation. Researchers have used language models to improve software engineering task such as code completion and code clone detection [1, 8, 18]. For compiler validation, DeepSmith [5], DSmith [24], and DeepFuzz [13] uses deep learning based sequence modeling to model the OpenCL and C languages from real world programs and found compiler bugs. All of these approaches target languages with complete available specifications while we target Simulink, which does not have such a specification publicly available.

The most closely related work DeepFuzzSL [21] use LSTM architecture to model Simulink. However they only train on synthetic models, citing the need for a larger training corpus. In contrast, we use a pre-trained language model and fine-tune it with open-source Simulink models.

Transfer learning for source code modeling is relatively new. Benito et al. studied the use of pre-trained models for source code generation and completion [4]. Hussain et al. proposed a transfer-learning based attention learner approach to improve code suggestions [11]. While earlier work focused on traditional languages we focus on a graphical CPS language.

6 CONCLUSIONS

Testing a commercial CPS development tool such as Simulink is hard as its codebase contains millions of lines of code and complete formal language specifications are not available. While deep learning techniques promise to learn such language specifications from sample models, deep learning needs a large number of training data to work well. SLGPT addressed this problem by using transfer learning, to leverage the powerful GPT-2 model that has been pre-trained on a large set of training data. SLGPT adapted GPT-2 to Simulink with both randomly generated models and models mined from open-source repositories. SLGPT produced Simulink models that are both more similar to open-source models than its closest competitor, DeepFuzzSL, and found a super-set of the Simulink development toolchain bugs found by DeepFuzzSL.

ACKNOWLEDGMENTS

The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper. Christoph Csallner has a potential research conflict of interest due to a financial interest with Microsoft and The Trade Desk. A management plan has been created to preserve objectivity in research in accordance with UTA policy. This material is based upon work supported by the National Science Foundation (NSF) under Grant No. 1911017 and a gift from MathWorks.

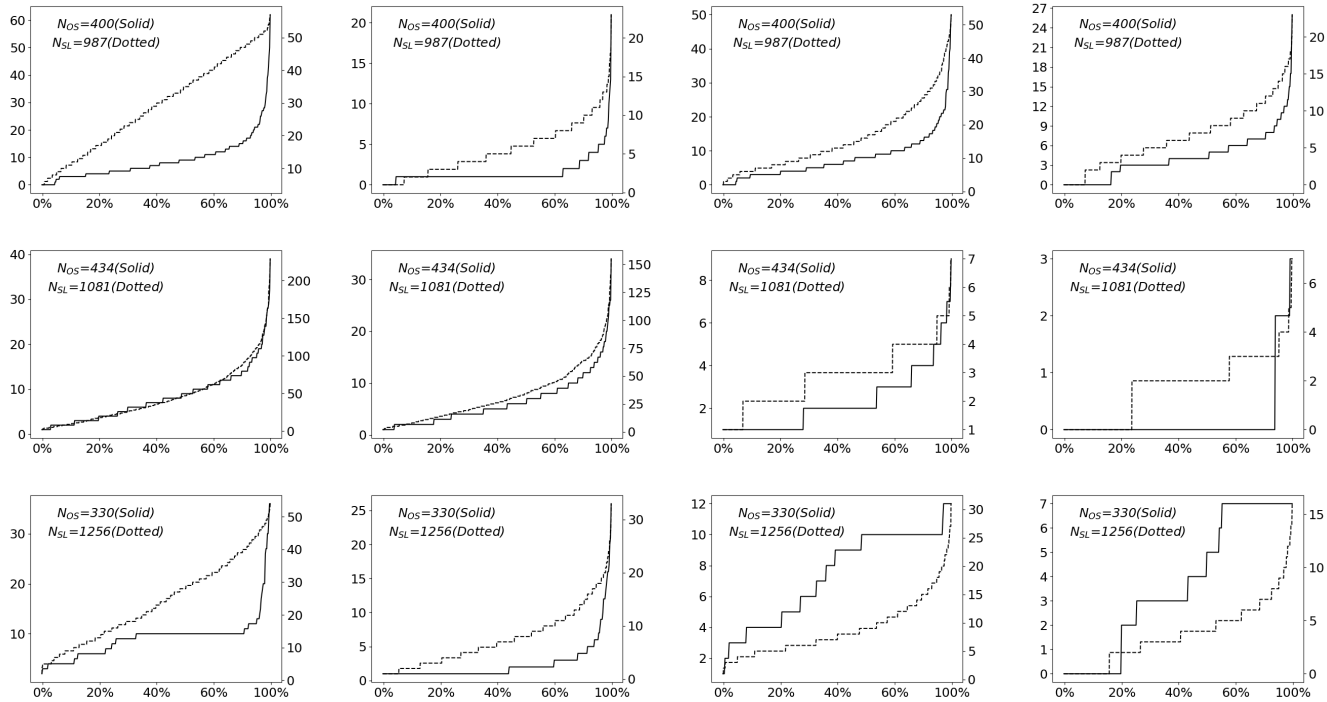


Figure 6: Training models (top), DeepFuzzSL-generated models (middle), and SLGPT-generated models (bottom). Left y-axis is for open-source training models and right y-axis is for SLforge-generated training models. X-axis is (valid) Simulink models sorted in ascending order for metric (from left to right column): Blocks per model, connected subgraphs, blocks in largest connected subgraph, maximum path length from a source to a sink block. Metrics of SLGPT-generated models are overall closer to the training models than DeepFuzzSL-generated models.

REFERENCES

[1] Qingying Chen and Minghui Zhou. 2018. A neural framework for retrieval and summarization of source code. In *ASE 2018*.

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

[3] Shafiul Azam Chowdhury, Sohil L Shrestha, Taylor T. Johnson, and Christoph Csallner. 2020. SLEMI: Equivalence modulo input (EMI) based mutation of CPS models for finding compiler bugs in Simulink. In *ICSE*. ACM, 335–346.

[4] Juan Cruz-Benito, Sanjay Vishwakarma, Francisco Martin-Fernandez, and Ismael Faro. 2021. Automated Source Code Generation and Auto-Completion Using Deep Learning: Comparing and Discussing Current Language Model-Related Approaches. *AI* 2, 1 (2021), 1–16. https://doi.org/10.3390/ai2010001

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

[6] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*.

[7] Sarah Fakhoury, Venera Arnaoudova, Cedric Noiseux, Foutse Khomh, and Giuliano Antoniol. 2018. Keep it simple: Is deep learning good for linguistic smell detection?. In *SANER'18*. IEEE, 602–611.

[8] Muhammad Hammad, Önder Babur, Hamid Abdul Basit, and Mark van den Brand. 2020. DeepClone: Modeling Clones to Generate Code Predictions. In *ICSR 2020*.

[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

[10] Jeremy Howard and Sebastian Ruder. 2018. Universal Language Model Fine-tuning for Text Classification. In *ACL 2018*. ACL, 328–339.

[11] Yasir Hussain, Zhiqiu Huang, Yu Zhou, and Senzhang Wang. 2020. Deep Transfer Learning for Source Code Modeling. *Int. J. Softw. Eng. Knowl. Eng.* 30, 5 (2020).

[12] MathWorks Inc. 2007. *Simulink® 7 Reference*. Chapter 9, Pg. 9–2 – Pg.9–10.

[13] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. 2019. DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing. In *Proc. 33rd AAAI Conference on Artificial Intelligence (AAAI)*. AAAI, 1044–1051.

[14] MathWorks Inc. 2021. *Block Libraries*. Retrieved April 27, 2021 from https://www.mathworks.com/help/simulink/block-libraries.html

[15] MathWorks Inc. 2021. *MATLAB & Simulink*. Retrieved April 27, 2021 from https://www.mathworks.com/products/simulink.html/

[16] Sinno Jialin Pan and Qiang Yang. 2009. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering* 22, 10 (2009), 1345–1359.

[17] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).

[18] Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *PLDI 2014*. 419–428.

[19] Romain Robbes and Andrea Janes. 2019. Leveraging small software engineering data sets with pre-trained neural networks. In *Proc. 41st International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER)*. IEEE.

[20] Sohil L Shrestha. 2021. *50417/SLGPT*. https://doi.org/10.5281/zenodo.4734223

[21] Sohil Lal Shrestha, Shafiul Azam Chowdhury, and Christoph Csallner. 2020. DeepFuzzSL: Generating models with deep learning to find bugs in the Simulink toolchain. In *DeepTest 2020*. ACM.

[22] TACC at The University of Texas at Austin. 2021. *Texas Advanced Computing Center - Homepage*. Retrieved April 27, 2021 from https://www.tacc.utexas.edu/

[23] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *30th Annual Conference on Neural Information Processing Systems*.

[24] Haoran Xu, Yongjun Wang, Shuhui Fan, Peidai Xie, and Aizhi Liu. 2020. DSmith: Compiler Fuzzing through Generative Deep Learning Model with Attention. In *2020 International Joint Conference on Neural Networks, IJCNN 2020*.

[25] Xi Zheng, Christine Julien, Miryung Kim, and Sarfraz Khurshid. 2017. Perceptions on the State of the Art in Verification and Validation in Cyber-Physical Systems. *IEEE Systems Journal* 11, 4 (2017), 2614–2627.