2023

Constructing Large Open-Source Corpora and Leveraging Language Models for Simulink Toolchain Testing and Analysis

Sohil Lal Shrestha

Constructing Large Open-Source Corpora and Leveraging Language Models for Simulink Toolchain Testing and Analysis

by

SOHIL LAL SHRESTHA

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON
December 2023

To my beloved Guddu

ACKNOWLEDGEMENTS

It takes a village to raise a child and more than a village to get a Ph.D. I have been blessed to have my educational journey enriched by great individuals who prepared me, inspired me, pushed me and even supported me financially to become the person I am today.

First and foremost, this dissertation would not have been possible without my advisor, Dr Christoph Csallner. His unwavering guidance, relentless support, and feedback have been paramount to all the research I have undertaken. His commitment to the quality of work is truly inspiring, and I hope that some of it has rubbed off on me over the years I have worked with him. I am also grateful to have collaborated with amazing researchers: Alexander Boll, Dr. Shafiul Azam Chowdhury, and Dr. Timo Kehrer.

I would also like to express my gratitude to Dr. Bahram Khalili, Dr. Won Hwa Kim, Dr. Deokgun Park for their interest in my research and for taking time to serve on my dissertation committee. I also want to express my thanks to Dr. Akshay Rajhans and Dr. Pieter Mosterman whose critical review of our manuscript significantly improved the quality of our research papers. I am thankful to all the anonymous reviewers whose constructive criticism of our work led to high-quality publications.

I am grateful to all the teachers that have taught me over the years. I am especially thankful to Dr. Rabindra Bista for teaching me the core concepts of computer science, which I hold close to my heart. I also want to thank my mentors and colleagues who offered guidance on pursuing graduate school. I am grateful to have

interacted with countless conference attendees, and such interactions often helped me gain a different perspective on the research problem I am working on.

I am grateful to my family for believing in me and encouraging me to achieve goals that, at the time, seemed unfathomable. I would like to thank my parents, Rameswor Lal Shrestha and Arati Shrestha and my extended family–Anar Shrestha, Arpana Shrestha, Anjana Shrestha, Rajan Shrestha, Arun Shrestha, Sushev Shrestha, Ashesh Shrestha–for their love and support. I am extremely fortunate to have my wife, Pratistha Shrestha, who supported me in every aspect of my doctoral studies. A special thanks goes to my sister, Aashara Shrestha, who supported me and helped me navigate life in a foreign country. I also want to thank Shuveksha Tuladhar for making living in a foreign country feel like a home away from home.

Lastly, I want to thank me. You did it !!!

November 8, 2023

ABSTRACT

Constructing Large Open-Source Corpora and Leveraging Language Models for Simulink Toolchain Testing and Analysis

Sohil Lal Shrestha, Ph.D.

The University of Texas at Arlington, 2023

Supervising Professor: Christoph Csallner

In several safety-critical industries such as automotive, aerospace, healthcare, and industrial automation, MATLAB/Simulink has emerged as the de-facto standard tool for system modeling and analysis, model compilation into executable code, and code deployment onto embedded hardware. Within the context of cyber-physical system (CPS) development, it is imperative to both rigorously test the development tools, such as MathWorks' Simulink, and understand modeling practices and model evolution. The existing body of work faces limitations primarily stemming from two factors: (1) contemporary testing methodologies often prove inefficient in identifying critical toolchain bugs due to a paucity of explicit toolchain specifications and (2) there exists a pronounced scarcity of a reusable and publicly available corpus of Simulink models for research.

In response to these challenges, we first pioneered the use of language models for random Simulink model generation by both training and fine-tuning (large) language models such as LSTM and GPT-2 on sample Simulink models. Second, we meticulously curated the largest collection of Simulink models: SLNET, which is

redistributable and contains detailed metadata. In addition, to encourage research on Simulink model evolution, we have curated EvoSL, a dataset of 900+ Simulink projects that has over 140k commits. Leveraging these datasets, we have systematically replicated previous studies, corroborating and/or refuting prior findings. As a further aid to the research community, we have developed ScoutSL, an open-source search engine for Simulink models. This tool simplifies the process of sampling Simulink projects from open-source domains, addressing the limitations of popular code hosting platforms that lack Simulink-specific filtering attributes. ScoutSL has already indexed over 100k Simulink models sourced from 18k projects.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

Introduction

MATLAB/Simulink is one of the popular toolchain for model-based design and it is widely used across a multitude of safety-critical domains [10–12]. The toolchain serves dual propose, primarily facilitating the prototyping of cyber-physical systems including design and simulation of such prototype. Moreover, it enables generation of embedded code through the prototypes meant for deployment to the target hardware. In light of this significance, it is important to develop measures to eliminate defects from the toolchain such as silent introduction of bugs in the executable due to incorrect compilation.

In software engineering, there are a number of ways to find bugs. For instance, one could try to formally verify the entire Simulink toolchain. Unfortunately, such formal methods require formal specifications which is not publicly available for MATLAB/Simulink toolchain, which can be partly attributed to the commercial nature of the tool. Even in the hypothetical scenario where such specification are available, formal verification of toolchain such as MATLAB/Simulink will be incredibly expensive as it spans across millions of lines of code and has a rapid release cycle occurring twice every year [13]. Also, toolchain testing suffer from test oracle problem similar to other software systems [14].

As formal verification is not feasible, prior works have explored in the direction of randomized differential testing [15–17]. Coined by McKeeman [18], randomized differential testing entails generation of random, valid test cases. These randomly generated test cases are then used to evaluate comparable programs, and any differ-

ences in the results may indicate the presence of bugs in one of the programs. In scope of Simulink toolchain testing, CyFuzz [17], first work that adapted randomized differential testing for Simulink, has a random model generation component which generates random, valid Simulink models which is then fed to its differential testing component. The differential testing component simulate (i.e. compiles, links and simulates) the model under different toolchain configurations [19–21]. Apart of handful of built-in library support, CyFuzz was inefficient in generating valid Simulink models as it initially generates models by randomly connecting blocks, often resulting invalid models and then iteratively addresses the issues until Simulink can successfully compile and simulate the models. The subsequent work, SLforge [22], leveraged free-form specification from the Simulink vendor's public webpage to guide the model generation that significantly reduce the time it took to generate valid models. On top of it, it also performed an exploratory study of 391 freely available Simulink models to prioritize the key model characteristics that realistic model is likely to have. The approach significantly improved the valid model generation rate as well as the efficiency of generation. SLforge built a parser to automatically incorporate the free form specifications to its model generator component but due to the parser's limitations, it could only collect parts of some specification. Furthermore, despite significant research and engineering investment, SLforge need to manually update the tool, whenever MathWorks updates model validity rules.

As SLforge is tightly coupled with Simulink, in this dissertation, we first proposed an alternative to random Simulink model generation by leveraging natural language processing and deep learning based methods. Random program generation for compiler testing using deep learning has gained some traction due to advent of neural language models where researchers have mainly focused on textual programming language such as C, OpenCL [23, 24] (these programming languages have specifications

publicly available). In natural language processing (NLP), language modeling is the task of predicting the next word (or tokens) conditioned in a set of previous words (or tokens). In the literature, there are series of advancement on how to learn language models starting from traditional n-grams models to current transformer based neural network architecture [25, 26] capable of capturing longer sequential structure of the text. In this dissertation, we first used Long Short Term Memory (LSTM) network to learn a language model directly from sample Simulink models (Chapter 2). The work, dubbed DeepFuzzSL [27, 28], reported high valid model generation rate i.e., 90% valid models on par with the SLforge, and found 2 confirmed bugs.

As the deep learning models typically require large number of training samples to generalize and no such large training corpus of Simulink models were readily available at the time, in DeepFuzzSL, we trained DeepFuzzSL with SLforge-generated Simulink models. Hence, DeepFuzzSL is inherently limited in terms of the validity rules it can learn from and generate. To overcome the limitation, the relatively nascent approach in NLP is transfer learning. Transfer learning involves pre-training the neural network on a task that has abundant training samples and then fine-tuning it in a related target task. The advent of transformer network accelerated the development of transfer learning based approach in NLP. Our SLGPT [29] (Chapter 3) tool used a pre-trained transformer based neural network, Generative Pre-trained Transformer-2 (GPT-2) [30] and finetune it on sample Simulink models. We also automatically mined open-source Simulink models to gather real world third party Simulink models to finetune GPT-2 in addition to using SLforge-generated models. In our experiments, SLGPT generated Simulink models were more similar to open-source models and it found a super-set of toolchain bugs DeepFuzzSL found.

The absence of corpus of open-source Simulink models has been a limiting factor hindering application of deep learning techniques. Also, understanding Simulink

modeling practices and its key characteristics has been critical to guide the random model generator in SLforge. Recognizing the dearth of a centralized corpus for the research community to do empirical study, Chowdhury et al. [31] manually collected over 1,000 freely accessible Simulink models. The corpus, while a step towards the right direction, suffer from issues such as scalability, inconsistencies, missing details such as metadata and were, at best, modest in size. Addressing the shortcomings of previous corpora, we built SLNET [32], which is 8X larger than prior corpora (Chapter 4). SLNET is fully self-contained including its metadata, redistributable and stored in Zenodo for long term access and storage. Further, we have automatically collected SLNET from open-source domain and thus, the accompanied collection tool can be used to enlarge SLNET as projects become available in open-source domains.

In a literature review, Boll et al. [33] found limited number of empirical Simulink research to be replicable in theory. They raised concerns about the low availability of research tools and experimental subjects (e.g, Simulink models), which severely hindered the replication and reproduction of existing work. To investigate such claims, we performed a thorough replication study of large-scale empirical research of Simulink on SLNET (Chapter 5). Our replication study highlighted several insights on documentation issues, incomplete replication packages and inconsistencies attributable to human error and bias. We also confirm and contradict several prior findings along providing key evidence on usefulness of dataset such as SLNET for research including model evolution.

One of the highlight in our replication study was the projects in SLNET can enable model changes or evolution study. However, the prior corpora, including SLNET, did not directly support such studies as they only contain a snapshot of the projects. Given much of the open-source projects are code dumps, it is challenging to get relevant projects for the evolution study. Thus, we curated EvoSL (Chapter 6), a

collection of 900 projects, consisting over 140,000 commits, mined using automated tools and scripts. Similar to SLNET, our new evolution study focused dataset is self-contained and redistributable. To further gauge the usefulness of corpus such as EvoSL, we replicated a study that analyzed the changes of closed-source industrial models. Our replication effort show that much of the original findings can be observed on the open-source models. We hope that both our dataset, SLNET and EvoSL, would allow the research community to develop hypothesis that can be tested easily than in an academic-industrial collaboration.

To further ease sampling of Simulink projects from open-source domains, we built a web based search engine, called ScoutSL [34] that allow users to search for relevant Simulink models using text-based as well as model and project metrics based attributes. ScoutSL has indexed over 100,000 Simulink models sourced from 18,000 Simulink projects. The search engine's key features is backed by the survey among Simulink researchers. ScoutSL is the first search engine specifically designed to search for Simulink models with key features not supported by any other search engines.

To summarize, the dissertation makes the following contributions:

- We present an alternative approach to generate random valid Simulink models where we directly learn from sample Simulink model files. To the best of our knowledge, DeepFuzzSL and SLGPT are the first work that employs LSTM and transfer learning to automatically generate Simulink models to test Simulink toolchain (Chapter 2 and 3).
- To encourage research on Simulink, we have curated two large dataset of Simulink models and projects, SLNET and EvoSL . Both the dataset and the tools used to build the dataset are open-source. We hope that our dataset would help research to test their hypothesis more easily than in an academic-industrial col-

laboration which may lead to faster progress and promote open science (Chapter 4 and 6).

- Our replication studies (Chapter 5 and 6) have uncovered several issues with existing empirical research while also providing evidence of usefulness of open source projects for various research.
- Our search engine, ScoutSL (Chapter 7), is first of its kind supporting search queries targeted towards Simulink attributes that may enable researchers to sample relevant Simulink projects for their research. Using a third-party projects/models in the research may reduce bias in the evaluation.

1.1 Author Contribution

The chapters in this dissertation are accepted publications. This section introduces the chapters along with the co-author contributions:

1.1.1 ***Chapter 2:*** DeepFuzzSL: Generating Simulink Models with Deep Learning to Find Bugs in the Simulink Toolchain

Proceedings: *2nd Workshop on Testing for Deep Learning and Deep Learning for Testing (DeepTest). 2020*

Authors: *Sohil Lal Shrestha, Shafiul Azam Chowdhury and Christoph Csallner*

Dr. Csallner supervised the entire project shaping the research direction, reviewing the research questions and experimentation setup. Shafiul Azam Chowdhury provided valuable feedback throughout the project and implemented one of the preprocessing step of DeepFuzzSL. I was responsible for designing and analyzing the experiments, and implementing core framework of DeepFuzzSL tool.

1.1.2 ***Chapter 3:*** SLGPT: Using Transfer Learning to Directly Generate Simulink Model Files and Find Bugs in the Simulink Toolchain

Proceedings: *25th International Conference on Evaluation and Assessment in Software Engineering (EASE). 2021*

Authors: *Sohil Lal Shrestha and Christoph Csallner*

Dr. Csallner supervised the entire project and provided continuous feedback developing the research questions and experimental setup. He also helped to improve the quality of paper. I was responsible for designing and implementing SLGPT along with running experiments and analyzing the results.

1.1.3 ***Chapter 4:*** SLNET: A Redistributable Corpus of 3rd-party Simulink Models

Proceedings: *19th International Conference on Mining Software Repositories (MSR). 2022*

Authors: *Sohil Lal Shrestha, Shafiul Azam Chowdhury and Christoph Csallner*

Dr. Csallner supervised the entire project from designing the corpus to denoising the corpus and strengthened the quality of the paper. Shafiul Azam Chowdhury analyzed the initial collection to remove license restricted projects. I was responsible for implementing tools for automatic collection of the corpus as well as metric extraction and analysis.

1.1.4 ***Chapter 5:*** Replicability Study: Corpora For Understanding Simulink Models & Projects

Proceedings: *17th International Symposium on Empirical Software Engineering and Measurement. 2023*

Authors: *Sohil Lal Shrestha, Shafiul Azam Chowdhury and Christoph Csallner*

Dr. Csallner provided valuable feedback that was helpful to shape the research questions and avoid experimental setup pitfalls. He significantly improved the paper's quality. Shafiul Azam Chowdhury provided valuable feedback throughout the project and helped answer one of the research questions. My contribution includes developing replication and analysis tools, analyzing the results and highlighting the key insights along with communicating with the author's of studies to ensure that the reproduction and replication are fair and consistent.

1.1.5 ***Chapter 6:*** EvoSL: A Large Open-Source Corpus of Changes in Simulink Models & Projects

Proceedings: *26th International Conference on Model-Driven Engineering Languages and Systems. 2023*

Authors: *Sohil Lal Shrestha, Alexander Boll, Shafiul Azam Chowdhury, Timo Kehrer and Christoph Csallner*

Dr. Csallner and Dr. Kehrer supervised the entire project and helped improve the quality of the paper. Dr. Csallner helped with formulating the research questions along with researching and reviewing centralized location to host EvoSL. Alexander Boll and Shafiul Azam Chowdhury analyzed the project's license file to ensure EvoSL is redistributable. Alexander Boll and I were involved in communicating with the original authors to ensure consistency in the replication we did. Besides communication, I was responsible for building tools to curate the dataset and its metrics along with analysis.

1.1.6 ***Chapter 7:*** ScoutSL: An Open-source Simulink Search Engine

Proceedings: *26th International Conference on Model-Driven Engineering Languages and Systems. 2023*

Authors: *Sohil Lal Shrestha, Alexander Boll, Timo Kehrer and Christoph Csallner*

Dr. Csallner and Dr. Kehrer supervised the entire project shaping the research direction and improved the paper. Alexander Boll conducted the user survey that included shortlisting relevant researchers, building survey questions, collecting input from them to visualize the survey responses. I designed and built the ScoutSL tool interface, its offline and online backend that collects as well as make the data available and deploy the tool in Amazon Web Services (AWS).

1.2 Other Publications and Presentation

1.2.1 ***ICSE Student Research Competition:*** Automatic Generation of Simulink Models to Find Bugs in a Cyber-Physical System Tool Chain using Deep Learning

Proceedings: *42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). 2020*

Authors: *Sohil Lal Shrestha*

1.2.2 ***ISSTA Doctoral Symposium:*** Harnessing Large Language Models for Simulink Toolchain Testing and Developing Diverse Open-Source Corpora of Simulink Models for Metric and Evolution Analysis

Proceedings: *32nd International Symposium on Software Testing and Analysis (ISSTA). 2023*

Authors: *Sohil Lal Shrestha*

1.2.3 ***Tapia Doctoral Consortium:*** Leveraging Language Models to Tackle Software Engineering Problems in Commercial Cyber-physical System Toolchain

Presentation: *2023 CMD-IT/ACM Richard Tapia Celebration of Diversity in Computing Conference (Tapia). 2023*

Authors: *Sohil Lal Shrestha*

CHAPTER 2

DeepFuzzSL: Generating Simulink Models with Deep Learning to Find Bugs in the Simulink Toolchain

This chapter was originally published in 2020 Virtual Workshop on Testing for Deep Learning and Deep Learning for Testing (DeepTest). It is reproduced here with permission without revision [27].

2.1 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 models. Existing randomized differential testing frameworks such as SLforge leverages semi-formal Simulink specifications to guide random model generation. This approach 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 a existing corpus of Simulink models. In our experiments DeepFuzzSL consistently generated over 90% valid Simulink models and also found 2 bugs in Simulink version R2017b and R2018b confirmed by MathWorks Support.

2.2 Introduction

Cyber-physical systems (CPS) are integration of cyberspace and physical world through a network of interconnected components such as actuators and sensors. En-

gineers typically prototype CPS with graphical block diagram using commercial development tools such as MathWorks Simulink [35] (a de-facto industry standard), which enable them to model, simulate and analyze their system. Furthermore, these toolchain can automatically generate embedded code that are often deployed in target hardware of safety critical systems. It is thus very important to find and remove bugs in such development toolchains.

In software engineering, there are a number of ways to find bugs. Ideally one can formally verify the entire Simulink toolchain, but it is not feasible due to its large and complex code base and lack of complete formal specification, which can be partly attributed to its commercial nature [16]. Like many other software systems, toolchain testing suffers from the test oracle problem [14].

An alternative is fuzzing, or random test case generation which is an effective way to identify bugs [36,37]. State-of-the-art Simulink-testing tool *SLforge* combined randomized fuzzing with differential testing and found 8 new bugs in Simulink [16]. Since Simulink does not have complete publicly available language specification, Chowdhury et al. [16] parsed semi-formal specifications from Simulink's web page automatically and rigorously incorporated them in SLforge's random model generator. While SLforge is proven effective, it inherently relies on documented specification to update it's random model generator.

To overcome the engineering effort of maintaining the tool with respect to subtle specification changes and adding new features while also preserving reasonable fidelity to the real world Simulink models, we propose to build a neural network model that can automatically generate Simulink models by learning directly from third-party Simulink models. We hypothesize that a neural network model should be able to capture undocumented Simulink specifications that is missed by earlier approach. The hypothesis is motivated by recent development in deep learning and natural

language processing research that have constructed probabilistic language models of how humans write code. Such approach have shown efficacy of random program generation without the need of rigorously defining rules or grammar in a random program generator [38, 39]. For e.g., *DeepSmith* [39], a deep learning based fuzzer, have reported 50+ bugs in OpenCL compiler such as LLVM and claimed that it can be easily extensible to other programming languages with minimum engineering efforts.

Earlier work on applying deep learning to compiler fuzzing have mostly focused on programming languages (such as C, OpenCL) whose complete specifications are publicly available. In contrast, we focus on Simulink that lacks complete specification making it a better candidate to validate language agnostic deep learning framework that earlier work claims [39].

In this work, we portray random Simulink model generation task as a language modeling problem (Section 2.3.1). Traditional statistical language model approach like n-grams fails to capture semantic relations, thus is not useful in our work. In contrast, neural language model (Section 2.3.1) captures the semantic and syntactic structure of a given language. While there are different types of neural network architecture (such as feed forward, convolutional, recurrent etc), we chose Long Short Term Memory(LSTM) [40], a variant of recurrent neural network, which has proven effective in language modeling [41].

In our DeepFuzzSL framework, we extend DeepSmith architecture to generate random Simulink models. In doing so, we verify their earlier claim and validate our hypothesis. In our preliminary evaluation, our trained DeepFuzzSL model is able to generate over 90% valid Simulink models and have found 2 bugs in Simulink versions R2017b and R2018b confirmed by MathWorks Support.

To summarize, this paper makes the following major contributions.

- To best of our knowledge, this is the first work that employs LSTM to automatically generate Simulink models to test the Simulink toolchain.
- In our experiment, DeepFuzzSL found 2 confirmed bugs in the widely used CPS development tool Simulink, one of which is missed by previous state-of-the-art.
- Our DeepFuzzSL prototype implementation and evaluation data are open source at GitHub [42].

2.3 Background

This section provides necessary information on neural language model, CPS models and commercial CPS tool chain Simulink.

2.3.1 Neural Language Model

Language modeling is the task of predicting the next word in a sequence based on the words already observed in the sequence. In essence, a language model assigns probability to a sequence of words, which is expressed as a joint probability over the words as:

$$P(w_1, w_2, \ldots, w_n) = P(w_1) \prod_{i=2}^{n} P(w_i | w_{i-1}, w_{i-2}, \ldots, w_1),$$

where w_i is the i-th word in a sentence of length n. So given a arbitrary word sequence $(x_1, x_2, \ldots, x_t)$, a language model can compute the probability distribution of the 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 $\mathrm{V} = \{w_1, \ldots, w_{|V|}\}$.

Conventional language models such as n-grams look at fixed *consecutive* context window (or finite window of *consecutive* previous words) to predict the next word. These kinds of language model couldn't be conditioned over large context window without running into *out of memory* issue [43].

On the other hand, a neural language model uses a neural network architecture to learn a language model as a distributed representation of words [25, 26]. Further improvement on neural language model gave rise to recurrent neural networks that can retain a state that can represent information from an arbitrarily long context window achieving state-of-the-art result in language modeling as well as other sequential learning task [44].

Using a language model, one can generate sequence of words conditioned on previous words. This is relevant to textual programming languages such as C, where, for example, a variable use never comes before variable definition. Although Simulink models are designed using graphical block diagrams, textual representation of the model follow the norm, where a block information never comes before the connection (or line) information making language model a good fit for this work.

2.3.2 CPS Model and Simulink

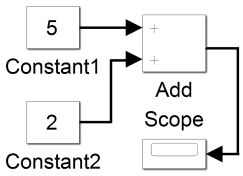


Figure 2.1: Example minimal toy Simulink model adding two constants, encoded in Listing 2.1.

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

Listing 2.1: Figure 2.1 model as text file (excerpt).

A CPS model is typically designed as a set of graphical *models* as seen in Figure 2.1. A model contains *blocks* that accepts data through the *input* ports. Block performs some operation on the data and can pass output to other blocks through *output* ports, using *connection* lines.

Simulink is a powerful, flexible, and de-facto standard commercial toolchain for CPS that supports various programming paradigms including data-flow and object oriented programming [45]. To design a CPS model, Simulink has support for various built-in block *libraries* [46]. Users can also create *custom-blocks* whose functionality can be defined through custom "native" code. Users can then *compile* and *simulate* a model. After compilation, Simulink offers different simulation modes [47]. In depth descriptions of CPS and Simulink can be found elsewhere [16, 48].

When a user attempts to open a Simulink model in Simulink, first the Simulink parser checks the model, possibly rejecting it and preventing the model from opening in Simulink. Once the model is opened, the user can compile and then simulate the model, which triggers different simulation phases [19]. In this paper, we consider a Simulink model to be *valid* if Simulink can open and compile the model without errors.

2.4 Overview and Design

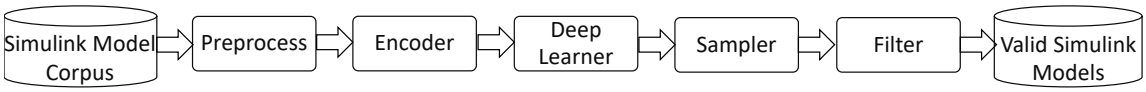


Figure 2.2: Overview of DeepFuzzSL's main processing phases.

DeepFuzzSL needs as input a set of seed Simulink models ("corpus"). Based on these seed models, DeepFuzzSL proceeds in five main processing phases, as shown in

Figure 2.2, to encode the seed models and use the encoded seeds to train a generative ML model, sample from the trained ML model, and decode the samples back to Simulink.

2.4.1 Seed Models: Simulink Model Corpus

We identified two main options for representing Simulink models in files, *mdl* and *slx*. While the main format since 2012 has been *slx*, this format has several drawbacks for our purposes. Specifically, *slx* stores a given model as a sequence of XML files, which is verbose and requires reasoning about a set of files.

In contrast, the earlier *mdl* format is also text based but more compact and thus easier to parse and generate for a deep learner. Each model is contained in a single *mdl* file and there is tool support for conversion between *mdl* and *slx*. DeepFuzzSL thus uses *mdl*.

While more compact than *slx*, *mdl* is still much too verbose for state-of-the-art deep learning systems. For example, the *mdl* representation of the Figure 2.1 toy example consists of over 1 kLOC with over 1,000 keywords and parameters following the structure shown in Listing 2.2. Before using a model as a seed for deep learning, we thus transform it as follows.

We remove *BlockDefaults* {...} and *AnnotationDefaults* {...} as such *mdl* file can be compiled without any issue.

We also observed information of model component desgined by user in Simulink are stored inside *System*{...}. Thus we stripped down all other model parameter such as configuration defaults, graphical interface defaults ensuring that the model can be compiled (aka valid).

```
Model {
  <Model Param_Name> <Model Param_val>
```

```
...
BlockDefaults {
  <Block Param_Name> <Block Param_val>
  ...
}
AnnotationDefaults {
  <Annotation Param_Name> <Annotation Param_val>
  ...
}
System {
  <System Param_Name> <System  Param_val>
  ...
  Block {
    <Block Param_Name> <Block Param_val>
    ...
  }
  Line {
    <Line Param_Name> <Line Param_val>
    ...
    Branch {
      <Branch Param_Name> <Branch Param_val>
      ...
    }
  }
  Annotation {
    <Annotation Param_Name> <Annotation Param_val>
    ...
  }
}
```

}

Listing 2.2: Typical *mdl* file format representation

2.4.2 DeepFuzzSL Processing Phases

Following are DeepFuzzSL's five main processing phases.

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.

The *mdl* representation lists all "Blocks{ ... }" first and then the connection between them later in the file. During our initial experiments, our trained deep neural net model was only able to generate Simulink models containing just blocks without any connection between them. To mitigate this issue, we interleave block and connection (or line) information in the *mdl* file such that every pair of connected blocks are defined first followed by their connection information.

Encoder: Neural network requires numeric sequences as inputs. Hence all seed models are converted to a sequence of fixed size feature vectors, where each integer is an index of predetermined vocabulary. We also studied different ways source code is encoded. In [38], character level encoding of source code is adopted. This minimizes the vocabulary size but leads to very long sequences. On the contrary, token level encoding leads to shorter sequences but increases vocabulary size as every literal is uniquely represented. To demonstrate the two extremes, we ran an experiment with 30 unmodified Simulink models' *mdl* files, each consisting of 5 to 15 blocks and then

extracted the number of tokens and vocabulary size in Table 2.1, with and without removing duplicate white space. To limit the size of the resulting feature vector, we encode Simulink models using a hybrid scheme that maps a few common keywords and parameters to tokens and the rest to characters.

Preprocess	Encoding	Tokens	Vocabulary
n/a	character	1,056,673	87
DWR	character	846,075	87
n/a	token	466,000	1,063
DWR	token	168,000	1,063

Table 2.1: Token count and vocabulary size of 30 Simulink models based on character and token level encoding; DWR = duplicate whitespace removal.

Deep Learner: We use a Long Short Term Memory (LSTM) network, a variant of recurrent neural networks following the success of many recent works [39, 49, 50]. We use a two layer LSTM network with 512 nodes per layer, which strikes a balance between the size of the neural net and the closeness of the learned distribution to the true distribution. This, in turn, yields a practical training time of the neural network. We defined the model using Keras [51] and Tensorflow [52] and open sourced the project on Github[1] for other researchers to train on their own corpus.

Sampler: After the training is complete, we seed the trained neural net with "Model {" tokens since every *mdl* file starts with it. Then we sample token-by-token to generate Simulink models. We halt the sampling when the opening and closing bracket counts become balanced or it reaches the maximum number of allowed tokens. Finally we decode the generated sequence back to text, which represents a Simulink model. Since we want to maximize the number of variations of generated models,

[1]https://github.com/50417/DeepFuzzSL/releases

we chose the next token from a randomized learned distribution, by performing a multinominal experiment.

Filter: Lastly we filter out the generated Simulink models by opening and compiling them in Simulink. The valid Simulink models can then be used to test Simulink for crashes or be used for differential testing.

2.5 Preliminary Evaluation

Deep learning requires a large number of seed models. While there is existing work on a public corpus of third-party open-source Simulink models [31], these third-party models are quite diverse. It would have taken us significant work to normalize these existing models, to bring them into a unified shape useful for our deep learning setup. To side-step these issues, we instead trained our LSTM network on 1,000 SLforge-generated models.

We performed the training remotely in the high performance Texas Advanced Computing Center (TACC) [53]. Specifically, we used TACC's Maverick 2 cluster, which has support for GPU accelerated deep learning research workloads. We ran our experiments on a single Maverick 2 GTX node[2], which has 128 GB RAM, two 8-core 2.1 Ghz Intel Xeon processors and 4 NVidia 1080-TI GPUs.

Using the Adam optimizer [54], we trained the network for 400 epochs using gradient descent with a learning rate of 0.002, decaying 5% every epoch with mini-batch size 64. We selected these hyper-parameters (epochs, learning rate, decay, batch size) based on the best result after multiple experiment runs. On TACC's Maverick2 GTX nodes, training the neural network took some 2 hours.

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

As a preliminary evaluation, at this stage we focus on if it is possible to build on LSTM-based deep learning an effective approach for finding bugs in the Simulink toolchain. Specifically, we explore the following two research questions.

RQ1 Can a LSTM-based deep learning approach generate valid Simulink models?

RQ2 Can a LSTM-based deep learning approach find bugs in the Simulink toolchain?

2.5.1 Generating Valid Simulink Models (RQ1)

To evaluate our approach, we sampled 1,024 Simulink models from our trained LSTM network, limiting the maximum number of tokens in each generated sample to 5,000 (since the largest seed model also had 5k tokens) and reported the ratio of valid Simulink models (i.e., models Simulink compiles without warning).

To encourage variation in the generated sample Simulink models, we adapted the following three sampling strategies [55]. These strategies either re-scale the probability or restrict the set of tokens to be sampled from.

1. Sampling with Randomization ("Temperature Sampling"): Temperature sampling allows to control the variability of the next generated token while preserving the fidelity of the corpus to the learned distribution. In temperature sampling, we increase or decrease the probability of the most likely next token before sampling it. Basically the probability of the next token is controlled by a hyper-parameter called temperature (T) as:

$$P(x^{t+1}|x^t \dots x^1) = \frac{P(x^{t+1}|x^t \dots x^1)^{1/T}}{\sum_{x=V} P(x^{t+1}|x^t \dots x^1)^{1/T}}$$

A low temperature (less randomization) makes the language model increasingly confident in its top choices while infinite temperature (full randomization) corresponds to uniform sampling.

2. Top-k Sampling: In top-k sampling we order the tokens by probability and select the top k tokens. With $p' = \sum_{x=V_k} P(x^{t+1}|x^t \dots x^1)$, the original distribution is re-scaled as

$$P(x^{t+1}|x^t \dots x^1) = \begin{cases} P(x^{t+1}|x^t \dots x^1)/p' & \text{if x} \in V_k \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

3. Top-p or Nucleus Sampling: Similar to top-k sampling, we select the highest probability tokens whose cumulative probability mass is greater than p. Specifically, the top-p tokens form the smallest set such that

$$\sum_{x=V_p} P(x^{t+1}|x^t \dots x^1) >= p$$

The probability distribution is re-scaled similar to Equation 2.1.

Sampling type	Valid model %
Sampling with randomization(T), T = 0.8	92.8
Top-k, k = 10	92.5
Top-p, p = 0.9	94.4

Table 2.2: Ratio of valid Simulink models generated via various sampling strategies. The randomization, k, and p values were chosen based on experiments with best results; p = cumulative probability.

In all three sampling strategies, we sample the next token based on a multinomial experiment with the given probability distribution. In our experiment, the sampling time for each sampling strategy took around 13 minutes.

Table 2.2 summarizes our results. Overall, in all cases we observed over 90% valid generated Simulink models. In other words, Simulink could compile over 90% of the DeepFuzzSL-generated models without warning. Nucleus sampling performed better than the other two, which aligns with earlier results [55].

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

Table 2.3: Summary of issue reports; *TSC* = *Technical Support Case* number from MathWorks; *O* = issue when opening model; *S* = issue when simulating model; *MW* = feedback from MathWorks; *K* = known bug; *N* = likely new bug.

2.5.2 DeepFuzzSL Found Bugs in Simulink (RQ2)

We encountered six Simulink crashes (triggered by six DeepFuzzSL-generated models), of which five crashes occurred while Simulink opened a model and one crash occurred while Simulink compiled a model (after successfully opening it). So far we have reported the latter issue plus one representative of the five "crash while opening" cases to MathWorks via its bug report website[3].

For each reported issue we received email from a MathWorks Support person who investigated the issue and tried to the find the crash's root cause. Unlike open source projects, MathWorks does not list all issue reports or even all confirmed bugs on its website. The bugs listed on their web site do not show their corresponding Technical Support case (TSC) number.

Table 2.3 summarizes the two issues we have reported. *MathWorks Support* has confirmed both issues as bugs. Following are the details of these two bugs.

2.5.2.0.1 TSC 03322011: Invalid Input Model (Known Bug) This DeepFuzzSL generated model consists of 3 discrete transfer function blocks. When trying to open this model Simulink crashes. Upon investigation, MathWorks Support determined that the generated model misses a certain parameter (OutputPortMap). Math-

[3]https://www.mathworks.com/support/bugreports/

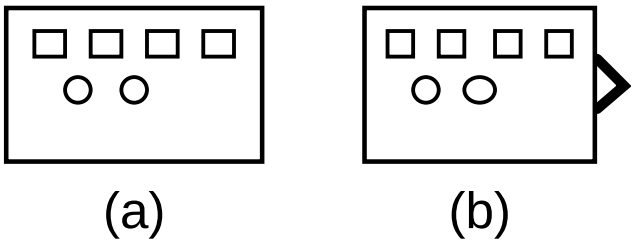


Figure 2.3: Signal Generator Block (a) DeepFuzzSL generated (b) from Simulink library

Works Support confirmed that this is a known bug. Instead of crashing, Simulink was supposed to produce an error and terminate normally.

2.5.2.0.2 TSC 03632450: Valid Input Model (Likely New Bug) This model generated by DeepFuzzSL consists of 25 blocks and 12 connections between them. Simulink could open this model normally without warning or errors. Simulink crashed when we tried to compile or simulate this model. In other words, DeepFuzzSL can generate models that pass Simulink's frontend parser.

Upon investigation, MathWorks Support provided as a reason that a signal generator block had a missing output port, which caused the crash. Figure 2.3(a) shows the signal generator block generated by DeepFuzzSL as opposed to one in Simulink library in Figure 2.3(b). MathWorks Support confirmed that this is a likely new bug. Instead of crashing Simulink should either produce an error or autofix the model. While the bug itself may be of low severity, it is an interesting one that validated our hypothesis "DeepFuzzSL can find bugs missed by SLforge". SLforge build a random model using Simulink block library, thus can not build Simulink models with such Signal Generator block.

2.6 Related Work

Fuzzing is a well established testing and validation approach. Many test case generators use programming language's grammar to systematically generate syntactically valid programs. Textual programming language such as Java, C, Rust have random program generator that aimed at finding bugs in their respective compilers [18, 56–58]. McKeeman present generators capable of enumerating programs from random ASCII sequences to C programs [18]. CSmith [59] is a widely known random C program generator which exploits infrequently combined language features to generate programs with clearly defined behavior but very unlikely functionality, increasing the chances to trigger a bug. Similar tool for other languages include JCrasher [56] for Java and Rust Typechecker Fuzzer [57] which uses constraint logic programming. Glade [58] generates programs after learning grammar from a corpus of example programs. Unlike our approach which learns the distribution, Glade uses the derived grammar to enumerate program uniformly at random.

While earlier work have largely focused on generating textual program (such as C , Java), limited work have been done for CPS models. CyFuzz [17] is perhaps the first tool to systematically generate Simulink models. As discussed throughout the paper, SLforge is the most closely related to our work. SLforge builds upon CyFuzz's limitations by incorporating informal Simulink specification into their random model generation. A subsequent work SLEMI [60] uses SLforge generated models to generate mutant of the seed model and found 9 confirmed bugs in Simulink. All of these work are tightly coupled with a particular CPS modeling language covering a subset of language specification and incurs high porting cost to other modeling language. In contrast, our work is loosely coupled with Simulink and has potential to cover undocumented specification.

Researchers are increasingly applying deep learning for software testing. Learn & Fuzz [61] learned from a corpus of PDF files to fuzz Microsoft Edge renderer. Closely related work DeepSmith [39] and DeepFuzz [38] learned probabilistic language model from a corpus of OpenCL and C programs and found multiple bugs in respective compilers. Both of the work target languages which have complete specification. On the contrary, although our work is built upon DeepSmith framework, we target CPS modeling language that does not have complete specification.

Similarly, while this paper looks for bugs in CPS tools such as Simulink using deep learning, a complementary line of work fuzz CPS models using machine learning and deep learning [62–66]. Liu et al. [65] use decision tree algorithm to stop test suite generation for fault localization of Simulink models. Chen et al. [62] use LSTM and Support Vector Regression to systematically guide generation of test suites for CPS network attacks. Their smart fuzzing system fuzzes actuator to drive CPS into unsafe state to diagnose cyber attacks. Kravchik et al. [66] study the use of convolutional and recurrent neural networks for detecting cyber-attacks in industrial control systems.

A summary of this work will also appear as a 2-page abstract in ICSE 2020's ACM Student Research Competition (SRC) [28]. In addition to the SRC summary, this paper adds details on sampling from the trained DeepFuzzSL model in Section 2.5.1 along with results in Table 2.2. Furthermore, this paper adds a description of the need for a hybrid encoding scheme with the results shown in Table 2.1. This paper also adds a *mdl* file structure and leverages the information while pre-processing that aid in training DeepFuzzSL in Section 2.4.1. This paper also includes details of the bug summary along with how we reported issues with MathWorks Support in Section 2.5.2.

2.7 Conclusions and Future Work

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 models. Existing randomized differential testing frameworks such as SLforge leveraged semi-formal Simulink specifications to guide random model generation which required 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 proposed to learn validity rules automatically by learning a language model. Our framework DeepFuzzSL learned from existing corpus of Simulink models and generated valid Simulink models. In our experiments DeepFuzzSL consistently generated over 90% valid Simulink models and also found 2 bugs confirmed by MathWorks Support.

Future work includes gathering a large Simulink model collection from public repositories such as Github and MathWorks File Exchange[4] and training the generative model on such a corpus as well as verifying the pre-processing heuristics.

[4]https://www.mathworks.com/matlabcentral/fileexchange/

CHAPTER 3

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

This chapter was originally published in 2021 ACM International Conference on Evaluation and Assessment in Software Engineering (EASE), Trondheim, Norway, 2021, pp. 260–265, doi: 10.1145/3463274.3463806. It is reproduced here with permission from ACM without revision [29].

3.1 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.

3.2 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 [27] 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, industrial automation and aerospace [67]. 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 [22] 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

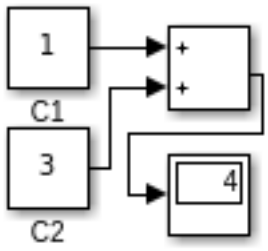
Simulink-specific training data to fine-tune such a pre-trained model for generating Simulink models.

Here, we fine-tune the Generative Pre-trained Transformer 2 (GPT-2) [30] 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 [68].

3.3 Background

Simulink [69] is a powerful commercial tool-chain for model-based design and has become a de-facto standard in several domains 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




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

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) [70]. Figure 3.1 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 [71]. 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.

3.3.1 Transfer Learning & NLP Language Models

Transfer learning [72] 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 [73] 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 [74]. GPT-2 uses transformer decoder as a building block and trains a language model on the WebText dataset [30]. Using transformers instead of LSTMs allows longer-range context capture. GPT-2's byte pair encoded vocabulary also supports Unicode (and

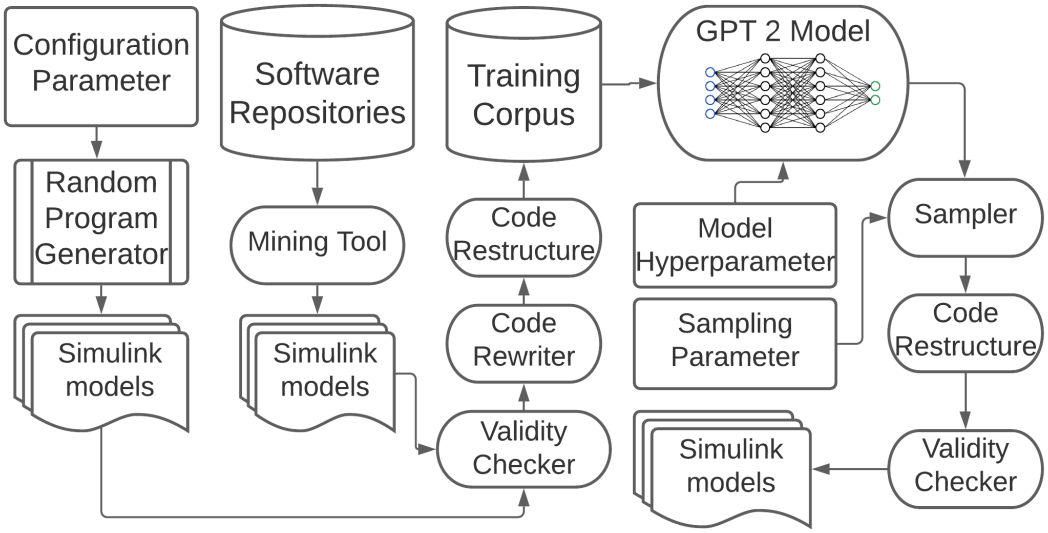


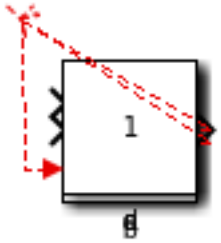
Figure 3.2: SLGPT obtains Simulink models from a random generator and open-source repositories, simplifies them, and uses them to adapt GPT-2 for finding Simulink crashes.

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.

3.4 Overview and Design

Figure 3.2 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.

[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  ... }...}}
```

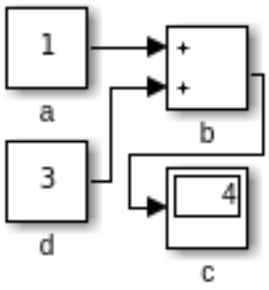
Figure 3.3: Figure 3.1 Simulink model and excerpt of its model file, after SLGPT simplified it to 45 lines, by removing layout info, restructuring code via BFS, etc.

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.

3.4.1 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




```
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 3.4: Figure 3.3 Simulink model and excerpt of its 45 line model file, after SLGPT restored it to Simulink-compliant style (plus manual layout changes for readability).

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.

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 ≠ ∅ or 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
```

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.

3.4.2 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 [75], 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 [76]). 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 3.1 example, if we assume Figure 3.3 shows a model produced by Algorithm-2, SLGPT then reorders its element definitions to the Simulink-friendly style of Figure 3.4.

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.

3.5 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 [53]. 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

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

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?

3.5.1 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 [22, 27, 60]). 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 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 3.6 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.

3.5.2 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.

3.5.2.1 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.

3.5.2.2 Case 04767975 (Known bug)

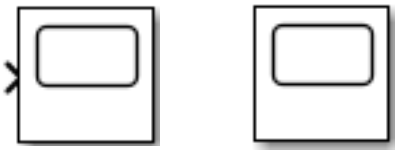


Figure 3.5: Scope (left) and Floating Scope (right).

Figure 3.5 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.

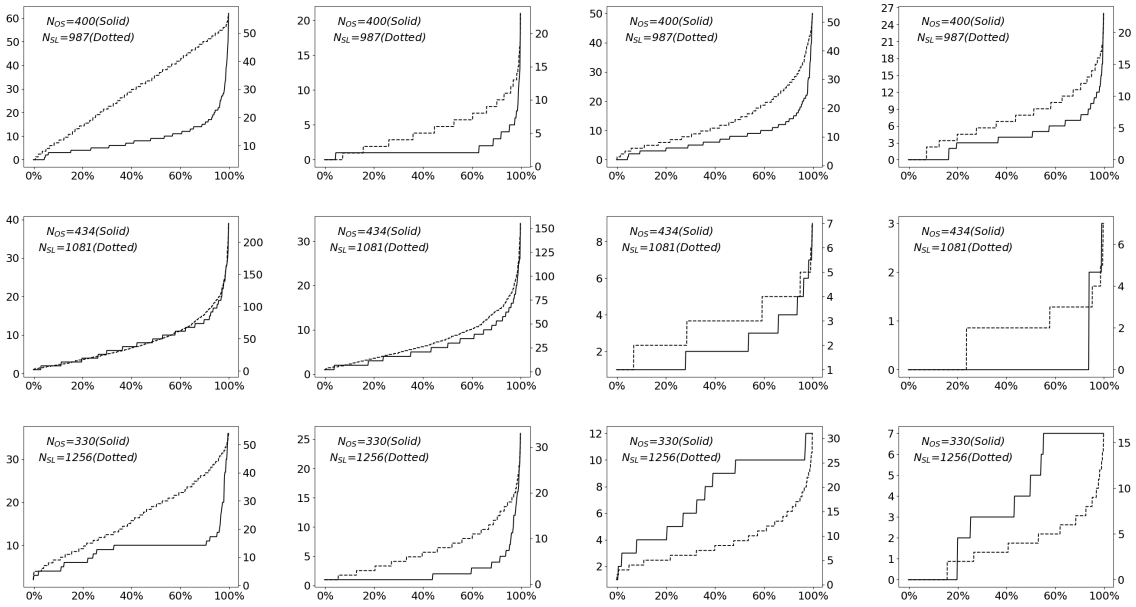


Figure 3.6: Training models (top), DeepFuzzSL-generated models (middle), and SLGPT-generated models (bottom). Left y-axis is for open-source training models and right y-axis is for SLforge-generated training models. X-axis is (valid) Simulink models sorted in ascending order for metric (from left to 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.

3.6 Related Work

Small training datasets are a common problem in deep learning applications. Researchers thus often use synthetic datasets [77]. Robbes et. al. showed a promising avenue to alleviate the dataset problem by using transfer learning [78], 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 [22]. Subsequently SLEMI generated semantic-preserving mutants of a seed model for differential testing of the Simulink toolchain [60]. 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 [79–81]. For compiler validation, DeepSmith [82], DSmith [24], and DeepFuzz [23] 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 [27] 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 [83]. Hussain et al. proposed a transfer-learning based attention learner approach to improve code suggestions [84]. While earlier work focused on traditional languages we focus on a graphical CPS language.

3.7 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.

CHAPTER 4

SLNET: A Redistributable Corpus of 3rd-party Simulink Models

This chapter was originally published in 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR), Pittsburgh, PA, USA, 2022, pp. 237–241, doi: 10.1145/3524842.3528001. It is reproduced here with permission from ACM without revision [85].

4.1 Abstract

MATLAB/Simulink is widely used for model-based design. Engineers create Simulink models and compile them to embedded code, often to control safety-critical cyber-physical systems in automotive, aerospace, and healthcare applications. Despite Simulink's importance, there are few large-scale empirical Simulink studies, perhaps because there is no large readily available corpus of third-party open-source Simulink models. To enable empirical Simulink studies, this paper introduces SLNET, the largest corpus of freely available third-party Simulink models. SLNET has several advantages over earlier collections. Specifically, SLNET is 8 times larger than the largest previous corpus of Simulink models, includes fine-grained metadata, is constructed automatically, is self-contained, and allows redistribution. SLNET is available under permissive open-source licenses and contains its collection and analysis tools.

4.2 Introduction

Currently there is no collection of Simulink models that is commonly used in empirical studies. Though there have been previous model collections, they lack fine-grained meta-information, are not self-contained, and are not redistributable due to restrictive or missing licenses—making them hard or impossible to use for most empirical researchers. Given the lack of such a collection, the few existing empirical studies of Simulink models have been limited to proprietary models or a small number of public models [22, 86, 87].

Deepening our understanding of Simulink models and modeling practices is important, as Simulink is a de-facto standard tool in several safety-critical industries such as automotive, aerospace, healthcare, and industrial automation—for system modeling and analysis, compiling models to code, and deploying code to embedded hardware [67, 88]. Having a large corpus of third-party Simulink models may make it easier for engineers and researchers to produce, reproduce, and validate empirical results about Simulink models, modeling practices, and tools that operate on such models.

The most closely related previous work has studied an initial collection of 391 third-party Simulink models [22] and later extended it to a curated corpus ("SC") of some 1k third-party Simulink models [31]. Boll et al. [89] collected an updated version of SC and assessed the corpus's suitability for empirical research. While pioneering larger studies and validating that models from such a corpus can be similar to industrial models, these collections consisted of a list of URLs to non-permanent resources [22] and contained models with unclear license information [31]. These collections were largely manual, which lead to inconsistencies (empty projects, duplicate projects, and missing metadata), relatively modest collection size, and may yield unintended human errors and bias.

To address these limitations, SLNET automates corpus construction and analysis, including data acquisition, cleaning (except for the rarely required manual review of a new license type), metric computation, and packaging. SLNET thereby automatically mines and analyses Simulink models from the two most popular repositories for sharing Simulink models, yielding a collection of thousands of models that is fully self-contained and allows redistribution.

To allow fine-grained selection of Simulink models and projects, SLNET computes several project-level and model-level metrics [89] and exposes them in a SQL database. SLNET similarly identifies and labels libraries and models that are test harnesses [90]. To summarize, this paper makes the following major contributions.

- SLNET is redistributable and 8 times larger than the prior largest known corpus of third-party Simulink models.
- SLNET [2] and its tools [91, 92] are available under permissive open-source licenses (CC BY and BSD 3-clause), e.g., SLNET is at: `https://doi.org/10.5281/zenodo.5259648`

4.3 Background on Simulink

Simulink [69] is a widely used commercial tool-chain for model-based design [67, 88][1]. Engineers typically design a cyber-physical system (CPS) model in Simulink's graphical modeling environment. A Simulink model such as Figure 4.1 is a *block diagram*, where each block represents equations or modeling components. Depending on the block type, each block can accept input (via input ports), perform some operation on its inputs, and produce output (via output ports), which then can optionally be forwarded to other blocks via explicit or implicit connection lines (aka

[1]Searching for "Simulink" jobs on LinkedIn in the US currently yields over 5k job postings: `https://www.linkedin.com/jobs/search/?keywords="simulink"&location=US`

signal lines). Simulink users can add blocks from various built-in *libraries* and toolboxes [71], and can also define *custom blocks* in "native" code (e.g., in C) using the S-function interface.

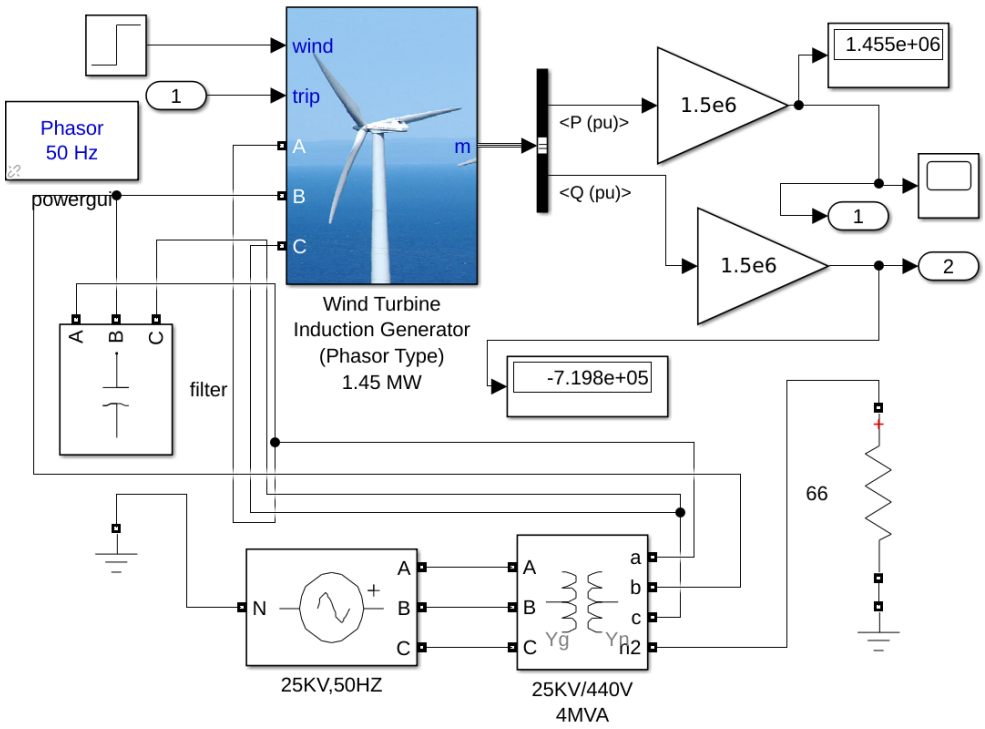


Figure 4.1: Sample SLNET Simulink model of a 1.5MW wind generation plant [1] with 18 blocks and 23 connections.

To deal with model size, users can create hierarchical models, by (recursively) grouping blocks in (a) a *Subsystem* or (b) in a separate model via *Model Reference*. Simulink does not permit a cyclic model hierarchy, but there may be block connection (aka data dependence) cycles, including algebraic loops[2].

As a first step, *compiling* translates the model into a toolchain-internal representation. When *simulating* the compiled model, the toolchain computes the output of each block at successive time steps over a specified time range using pre-configured

[2] `https://www.mathworks.com/help/simulink/ug/algebraic-loops.html`

numerical solvers. *Fixed-step* solvers solve the model at fixed time intervals whereas *variable-step* solvers automatically adjust the time intervals at which the model is solved. Simulink may reject a model if it cannot numerically solve an algebraic loop. Simulink offers different simulation modes, i.e., *Normal* mode "only" simulates blocks, *Accelerator* speeds up simulation by emitting native code, and *Rapid Accelerator* produces a standalone executable[3].

4.4 SLNET Design & Construction

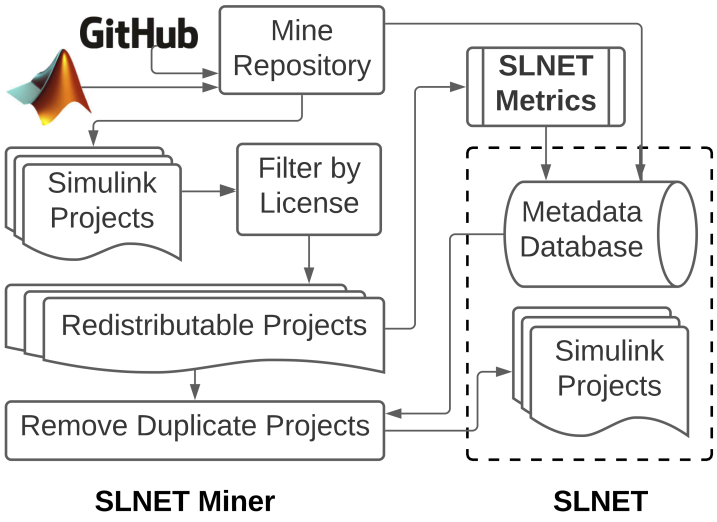


Figure 4.2: Overview: SLNET-Miner collects files and data, removes empty and duplicated projects or those without appropriate license. SLNET-Metrics extracts model metrics.

SLNET is not a superset of earlier Simulink corpora [31, 89] as earlier corpora were neither self-contained nor redistributable. Figure 4.2 gives an overview of SLNET's construction. We built SLNET from models shared in GitHub [93] and

[3]Simulink's embedded code generation workflow for deployment on target platforms is distinct from these simulation modes.

MATLAB Central [94]. Due to time limitations we do not collect Simulink models from smaller repositories such as GitLab [95] and SourceForge [96]. Before removing projects that are empty, duplicate, or have an unclear license, a quick search for "Simulink" yields some 60 GitLab and some 70 SourceForge projects.

While GitHub offers commit-level version control, MATLAB Central "only" serves project releases. To limit SLNET's size and due to the different versioning (git commits vs. project releases), in February 2020 we "only" collected Simulink project snapshots (i.e., all current project files plus project metadata).

GitHub provides a REST API to discover projects and extract them with their metadata. SLNET-Miner queries the GitHub API (via PyGithub [97]) with the keyword "Simulink". Unlike previous work [22, 31], we used keyword search and not file extension search, as file extension search is typically intended to search within a given GitHub repository and using file extension search in GitHub's search page produced many false positives.

The GitHub API expose 23 types of project-level information [98], of which SLNET retains 20. The other 3 are redundant (full project name) or API-internal (API query relevance score and node id). From the API we also obtain each project's topics (user-created labels and tags). From the downloaded project files, we extracted the list of Simulink model files plus the project's license.

As MATLAB Central "only" offers an RSS feed [99] for its file exchange platform, we filter the search result feed by Simulink models and then parse the feed to collect each project's download URL plus 14 other types of project metadata. Since from the RSS feed we could not construct the download URL for all projects, we extracted 2,941 of the 3,110 available projects.

4.4.1 Data Cleaning & Storage: ZIP + SQLite

We remove projects without Simulink models (i.e., file extensions *slx* or *mdl*) and projects we know to contain synthetic models (i.e., model generators [22, 60]). We heuristically search for other model generators (via terms "automat", "random", "fuzz", and "generate") in project titles, project descriptions, and project tags, which yielded 530 projects (e.g., on fuzzy logic). As we did not find evidence that these projects generate models we kept them in SLNET.

Table 4.1: Data cleaning: Real = has 1+ models (likely non-synthetic); License = has a license; SLNET+D = license allows redistribution; SLNET = has a model with 1+ blocks after removing potential duplicate projects; Model counts here include 1,130 library and 9 test harness models.

	Projects				Models
	Real	License	SLNET+D	SLNET	SLNET
GitHub	1,284	232	231	225	2,088
MATLAB Cl	2,941	2,746	2,728	2,612	7,029
Total	4,225	2,978	2,959	2,837	9,117

We then remove projects without a license or whose license does not allow redistribution. GitHub has a structured way for authors to set a license, which GitHub converts to a file (and exposes via an API). We manually reviewed the remaining 50 projects' licenses (where GitHub did not understand the author's license or for MATLAB Central projects without a BSD license).

We heuristically remove potentially duplicate projects. We consider project A a duplicate of B if (1) A and B contain the same number of Simulink model files and (2) there is a bijective mapping between models in A and B based on our Section 4.4.2 model metrics (excluding compile time). If A and B are from the same data source (GitHub or MATLAB Central), we keep the first-created one in SLNET. Otherwise,

we keep the one from GitHub, as it offers more fine-grained meta-data. Finally, we remove dummy projects (projects whose Simulink models all have zero blocks).

Table 4.1 summarizes data cleaning. After removing model generators we downloaded 4,225 projects with at least one Simulink model, of which 2,978 had a license, of which 2,959 allowed redistribution. Removing 112 potentially duplicate plus 10 dummy projects yielded 2,837 projects and their 9,117 Simulink models in SLNET.

SLNET is on Zenodo (a second archive contains the 112 duplicate projects) [2]. Each project has a snapshot of its files in a ZIP archive in either the GitHub or MATLAB Central directory. Each project is named **ID.zip**, where ID is an identifier defined by GitHub or MATLAB Central. SLNET includes the Figure 4.3 SQLite[4] database. It contains project-level information (license type, etc.) from the source repositories and the model metrics our tools extracted. Users can thus select models and projects from SLNET via SQL queries.

4.4.2 Project & Model Metrics

Table 4.2: SLNET's project engagement distributions are long-tailed as in other studies of open-source projects [6–9].

	Metadata	Min	Max	Avg	Med.	SD
GitHub	Stargazers	0	128	3.5	0	12.1
	Forks	0	122	2.8	0	10.7
	Open Issues	0	82	1.2	0	6.5
MATLAB Cl	Comments	0	218	3.5	1	12.3
	Ratings	0	108	2.9	1	6.8
	Avg. Rating	0	5	2.5	3	2.2

[4]SQLite is widely used, free, self-contained, server-less, zero-configuration, backwards compatible, and cross-platform: `https://www.sqlite.org/index.html`

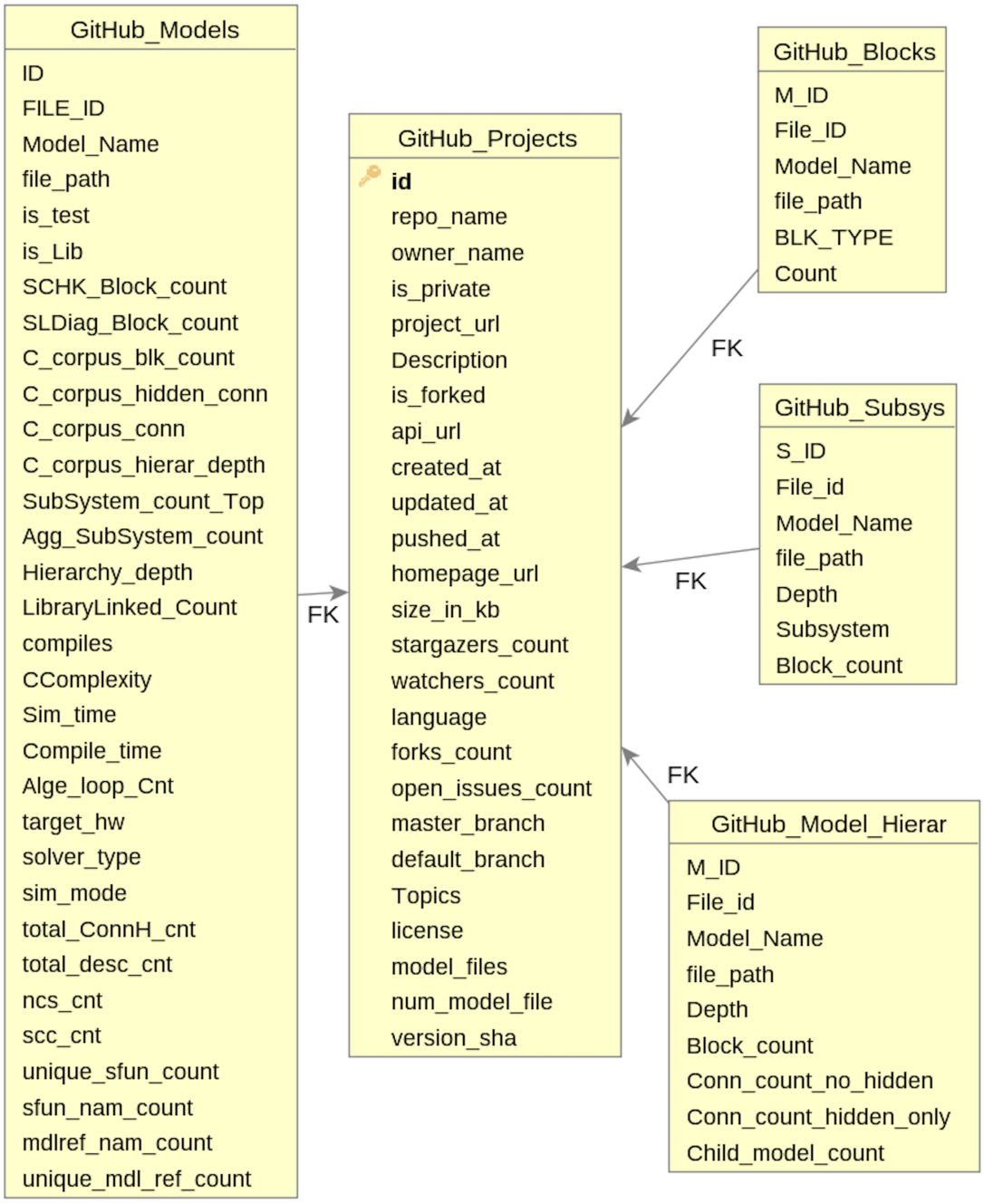


Figure 4.3: SLNET database schema (GitHub portion). The MATLAB Central portion only differs in its _Projects table [2].

To get an insight into the projects' domain and popularity we first searched the user-generated project tags (i.e., GitHub "topics" and MATLAB Central "categories") for common domains (i.e., the Simulink project domains identified by Boll et al. [89]), yielding Electronics (983), Automotive (64), Communications (61), Robotics (52), Energy (48), Aerospace (47), Biotech (20), and Medicine (2). Table 4.2 shows data often used as proxies for project popularity or engagement (e.g., people who have starred or forked a GitHub project or provided a 1–5 star rating for a MATLAB Central project). For example, a SLNET GitHub project has on average 2.8 forks.

	Models		Hierarchical		Blocks		Connections		Solver Step		Simulation Mode			
Source	M	Mc	Mh	Mh^{t0}	B	B^{t0}	C	C^{t0}	Fixed	Var	Nor	Ext	PIL	Ac
GitHub	1,639	541	878	1,304	190,321	414,241	188,285	395,725	860	762	1,501	103	2	14
MATLAB Cl	6,251	3,636	3,893	5,566	838,956	3,197,221	915,975	3,084,605	1,757	4,493	5,984	186	2	76
Total	7,890	4,177	4,771	6,870	1,029,277	3,611,462	1,104,260	3,480,330	2,617	5,255	7,485	289	4	90

Table 4.3: SLNET's model metrics after removing library & test harness models; M = models; Mc = models we could readily compile; Mh = hierarchical models (readily compilable and otherwise); C = non-hidden connections; ^{t0} = via SC's metric tool; Var = variable; Nor = normal; Ext = external; PIL = processor in the loop; Ac = accelerator. For 18 models the API did not indicate simulation mode or solver type. The remaining 4 models are configured for Rapid Accelerator simulation mode.

To extract commonly used model metrics (such as number of blocks, connections, subsystems, and linked blocks[5]) we implemented the SLNET-Metrics tool [92] on top of Simulink's APIs. While our Simulink installation and toolbox configuration [100] cannot compile a significant portion of SLNET models (mostly due to missing toolbox licenses), these APIs still compute metrics for these non-compiling models, except for three metrics (algebraic loops, cyclomatic complexity, and compile time).

[5] `https://www.mathworks.com/help/simulink/ug/creating-and-working-with-linked-blocks.html`

SLNET-Metrics failed to compute metrics for 88/9,117 models (21 from GitHub, 67 from MATLAB Central). Most of these 88 were due to Simulink version issues (missing Simulink toolboxes, model name conflicts with a keyword or toolbox file name) and bugs introduced by manually-edited model files. SLNET does not include metrics for these 88 models and thus also ignores them for the above duplicate-via-bijection removal.

SLNET-Metrics collects each model's hierarchical depth, solver type, simulation mode, target hardware, and use of S-functions and model references. While SLNET models contain elements from the state-machine toolbox Stateflow, Stateflow is out of scope and our metrics do not count the Stateflow-contents of a Simulink block.

Unlike SC, SLNET-Metrics does not count nested blocks imported from libraries or their connections (aka "masked subsystems"). This mirrors procedural code metrics, which also do not count LOC a program imports from a library. As SC's counting of such imported blocks approximates the model's overall conceptual complexity [101], Table 4.3 also includes these counts. As an example, the Figure 4.1 model imports blocks from the Simscape toolbox, yielding a SC-style block count of 907 with 919 connections.

The Simulink API labels only 9 SLNET models as a test harness, likely because many open-source projects do not have the required "Simulink Test" license to develop such tests. Beyond this official classification SLNET contains likely "work-around" test harnesses. The SC metrics tool heuristically matches model and folder names with "test" and "harness" and SLNET labels such models separately.

We performed sanity checks on the model metrics other papers reported about industrial models (block count, etc.). We also randomly sampled from the top 100 largest models in SLNET. Based on the sampled models' documentation we are confident that these were real human-created (non-synthetic) models.

4.5 Potential Research Directions

Since most industrial models are proprietary SLNET is unlikely to reflect their distribution. Instead, the goal is to provide the largest possible redistributable self-contained corpus of non-synthetic models. Different research projects will require different SLNET subsets (e.g., many small models for training deep-learning classifiers vs. large models to evaluate a technique's scalability), which the SQL metadata database facilitates. Having more models is better, especially in deep learning, but also when trying to understand the breadth of modelling practices, or when looking for edge cases (e.g., to test model analysis tools). Following are example directions.

While there has been significant interest in other software engineering areas [102–104], applying machine learning is relatively under-explored in model driven engineering [105, 106]. To work well, many machine learning and deep learning algorithms require large training sets. SLNET with its many models and rich metadata is thus well-suited. For example, a SLNET subset has been used to train a deep learning model for random Simulink model generation, to find bugs in the Simulink toolchain [29]. Due to their smaller size, this would have not been possible with the earlier corpora.

Due to the lack of easily available open-source models that fit certain characteristics, recent work reverted to evaluating tools on synthetic models [60]. SLNET offers a complimentary (and often preferred) evaluation option with human-authored models.

Recent work including in clone detection, refactoring, model slicing, and model smells has relied on evaluations with few proprietary Simulink models [107–111]. For example, Deissenboeck et al. [107] evaluated their clone detection approach on a single proprietary Simulink model with 20k blocks. Complementing such evaluations with

a variety of open-source models from SLNET could make such studies more general and easier to replicate.

Understanding modeling practices would enable researchers to tune their tools to how engineers use Simulink in various settings. For example, SLforge guides its random model generation by how often blocks appear in 391 open-source models [22]. The larger size of SLNET could thus, e.g., yield useful insights for tool design.

There may also be interesting correlations between metrics, maybe connecting model metrics to project metrics (e.g., model size metrics with project engagement). More generally, SLNET could contribute to a deeper understanding of model modularity, comprehension, quality, and maintainability [112–115].

While SLNET is unlikely to exactly represent closed-source development, the precise shape of this relation is an open question. For example, for the related domain of Object Constraint Language (OCL) expressions [116], Mengerink et al. found the distribution of expression complexity mined from GitHub projects reflects the distribution in closed-source projects, so open-source projects can be used as a proxy for industrial projects [117, 118].

4.6 Threats to validity

Due to its search heuristics SLNET-Miner may miss Simulink models (e.g., by missing some of the non-documented RSS feed URLs). Furthermore, since SLNET contains only redistributable projects, results may not be representative of all open source Simulink projects. On the flip side, while removing forks and duplicates, SLNET likely contains clones (from near-duplicate projects to adapted model portions), which can be an opportunity for clone-based research (and a challenge for others). Finally, SLNET-Metrics calls the Check API of Simulink R2019b. While

this API has been available since Simulink R2017b, its behavior may change across releases and thus yield different metric values in future Simulink versions.

4.7 Conclusions

In conclusion, SLNET is the first self-contained and redistributable corpus of freely available third-party Simulink models that aims to facilitate future empirical studies on model based design. SLNET has several advantages over earlier collections. Specifically, SLNET is 8 times larger than the largest previous Simulink corpus, includes fine-grained metadata and is constructed automatically. SLNET is available under permissive open-source licenses and contains its collection and analysis tools.

CHAPTER 5

Replicability Study: Corpora For Understanding Simulink Models & Projects

This chapter was originally published in 2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), New Orleans, LA, USA, 2023, pp. 130–141, doi: 10.1109/ESEM56168.2023.10304867. It is reproduced here with permission from IEEE without revision [119].

5.1 Abstract

Background: Empirical studies on widely used model-based development tools such as MATLAB/Simulink are limited despite the tools' importance in various industries.

Aims: The aim of this paper is to investigate the reproducibility of previous empirical studies that used Simulink model corpora and to evaluate the generalizability of their results to a newer and larger corpus, including a comparison with proprietary models.

Method: The study reviews methodologies and data sources employed in prior Simulink model studies and replicates the previous analysis using SLNET. In addition, we propose a heuristic for determining code-generating Simulink models and assess the open-source models' similarity to proprietary models.

Results: Our analysis of SLNET confirms and contradicts earlier findings and highlights its potential as a valuable resource for model-based development research. We found that open-source Simulink models follow good modeling practices and contain models comparable in size and properties to proprietary models. We also col-

lected and distribute 208 git repositories with over 9k commits, facilitating studies on model evolution.

Conclusions: The replication study offers actionable insights and lessons learned from the reproduction process, including valuable information on the generalizability of research findings based on earlier open-source corpora to the newer and larger SLNET corpus. The study sheds light on noteworthy attributes of SLNET, which is self-contained and redistributable.

5.2 Introduction

There are only a few empirical studies of open-source MATLAB/Simulink artifacts, maybe due to a widespread perception that open-source Simulink artifacts are typically small, do not represent closed-source development, and are often hard to acquire [22,60,120–122]. Most empirical Simulink studies to date have instead relied on academic-industry collaborations—to get access to large closed-source Simulink artifacts [123]. Most empirical results on Simulink development and artifacts are thus based on case-studies of closed-source artifacts that (even when providing detailed experimental design descriptions and measurement tools) are hard to reproduce or replicate [33].

It is well-known how important replication is for scientific progress. Successful experiments need to be cross-validated under different conditions before they can be considered a part of science and interpreted with confidence [124]. Working towards large open-source Simulink corpora and empirical results that are easier to reproduce and replicate are thus important goals, given how widely Simulink is used in industry in safety-critical domains such as automotive and healthcare.

Towards these goals, recent initial work created via manual mining a first large corpus (which we call SC [31]) of open-source Simulink models and investigated mod-

eling practices on a re-collected version of that corpus (SC_{20} [89]). The work found that some of these manually-collected Simulink models are suitable for empirical research, based on model metrics analysis and a qualitative assessments by a domain expert [89]. Follow-up work automated Simulink model collection, yielding the larger SLNET corpus that also allows redistribution [32]. However we are not aware of earlier work that either characterizes this larger SLNET corpus or uses it to replicate earlier empirical studies of Simulink models.

We thus first reproduce studies that are based on the initial SC large-scale Simulink model corpus, identifying inconsistencies in the original studies. We then replicate results of the earlier studies using the newer and larger SLNET corpus. By re-running the original study designs, we found inconsistencies between the experimental results and the ones reported in the paper, attributable to oversight and incomplete documentation. Our replication study using SLNET confirmed several previous findings, such as the low utilization of model references and algebraic loops. In contrast to prior work, we only found a weak correlation between cyclomatic complexity and other model metrics. To summarize, this paper makes the following major contributions.

- Through empirical data, we identify inconsistencies in earlier empirical Simulink studies.
- We characterize the SLNET corpus in relation to earlier corpora of open-source Simulink models.
- On SLNET we replicate previous studies, which both confirms and contradicts earlier findings.
- We collect and distribute 208 SLNET git repositories, containing 9k+ commits including 5k model versions, as artifacts that can be analyzed by the community [125].

- Our analysis tool [126] as well as reproduction and replication data [125] are open-sourced and available.

5.3 Background

Using Simulink's graphical modeling environment, engineers can design a complex system model as a hierarchical *block diagram* [69]. Each block represents a dynamic system that may take input through its *input ports* and produce output via its *output ports*, either continuously or at specific points in time. A block can be from a Simulink built-in library [71], from a separate *toolbox* library, or a custom *S-function* block defined via "native" code (e.g., in C). Blocks pass data to each other via directed *connections* (aka lines). Simulink is a commercial de-facto standard toolchain in several domains such as aerospace, automotive, healthcare, and industrial automation.

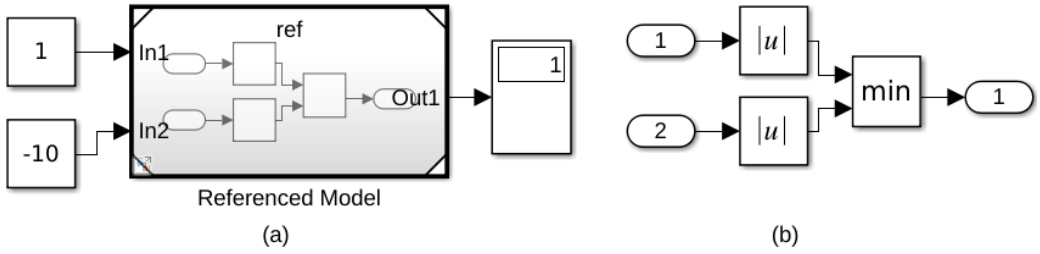


Figure 5.1: (a) A tiny Simulink example model, (b) shows the contents of (a)'s referenced model.

Simulink offers several hierarchy mechanisms, ranging from a *subsystem* block grouping that can only be used in one context to a *model reference* (which essentially calls an independent model via its own well-defined interface and can thus be widely reused) [127]. These constructs allow further recursive decomposition, enabling deeply nested models. Figure 5.1(a) shows a tiny example hierarchical model

that contains a model reference to the Figure 5.1(b) referenced model. Alternatively, the user can use library-linked blocks [128], that are references to blocks defined in a custom library [129], that enables reusability and centralized maintenance of block functionality across multiple models.

A *compiled* model can be simulated, where Simulink successively computes the output of each block over a specified time range using pre-configured numerical *fixed-step* and *variable-step* solvers. In an *algebraic loop*, a block's output can reach its input port in the same simulation step (i.e., without passing through a delay block), which complicates simulation. Besides *normal* mode, Simulink offers various *accelerator* modes to speed up simulation [130]. With additional toolboxes [131], from the model the user can then generate and deploy low-level code to the target hardware.

5.3.1 Simulink Modeling Guidelines and Best Practices

The MathWorks Advisory Board (MAB) is a group of commercial MathWorks customers that (starting with Daimler, Ford, and Toyota in 2001) publishes guidelines and best practices on developing and maintaining Simulink models. Besides standardization, these guidelines address key software engineering challenges such as creating models that are well-defined, readable, easy to integrate, and reusable.

In their current 2020 version [132] these guidelines include to (1) avoid algebraic loops as they are hard to simulate and cannot be compiled to target hardware, (2) use S-functions to implement custom algorithms, (3) use subsystems to modularize the model by functional decomposition, and (4) use model references to create hierarchies of reusable components.

5.3.2 Cyclomatic Complexity & Size Metrics in Simulink

McCabe introduced cyclomatic complexity and argued it corresponds to our intuitive notion of complexity. McCabe also reported on a set of 24 Fortran subroutines with "high" (>10) cyclomatic complexity. The subroutines' ranking by cyclomatic complexity closely correlated with their ranking by reliability [133]. With some 9k citations this article has been highly influential in academic software engineering.

Some five decades later the question of measuring program complexity and program understanding remains an active research area with several recent advances [134–136]. Researchers keep returning to cyclomatic complexity with recent tweaks [137, 138] and more fine- grained measures [139]. An example controversy was if cyclomatic complexity is just a proxy for program size (e.g., lines of code in a Java- or C-like language) [140], with recent empirical data showing cyclomatic complexity to remain independently valuable [141].

For Simulink, recent work has shown the value of size metrics (i.e., block count), e.g, metric outliers yield interesting findings [142]. Such results are also eventually reflected in industry practices. For example, while the MAB industry board's 2001 Simulink guidelines did not yet mention size metrics, the current 2020 version contains a recommendation ($\leq$60 LOC / function) [132]. However neither MAB guideline version mentions McCabe or cyclomatic complexity yet.

For calculating a metric, Simulink basically first flattens a given model into a single hierarchy level, essentially "inlining" both subsystems and referenced models. So if two blocks in a model refer to the same referenced model, for metric calculations the referenced model will appear in the flattened model twice. Simulink has an option to also similarly (recursively) inline the contents of (any) library blocks and prior work is split on activating this option when reporting metric results.

While a block diagram does not represent a procedural language's control-flow graph, Simulink still has several block types that provide control-like functionality. For example, the value a multiport-switch block receives on its first input port selects which of the remaining input ports the block will forward to its output port [143] (which corresponds to a procedural switch or nested if construct). Simulink thus first defines the cyclomatic complexity of each built-in block as the number of the block's conceptual branching decisions (i.e., mostly zero or one) and then sums up the cyclomatic complexity of all blocks in a given (flattened) model [144].

5.3.3 Scope of Empirical Studies of Simulink Models

The limited availability of repositories with large numbers of freely accessible Simulink models has restricted empirical studies that seek to understand Simulink model characteristics and metrics [113, 145, 146]. For example, Dajsuren et al. [112] investigated model metrics including cohesion and coupling using small subset of Simulink models.

Open-source Simulink models are generally considered insufficient to meet the high industry standards required for meaningful results [33]. To address this issue, Altinger et al. [147] published metrics from three proprietary Simulink models for researchers to analyze. However, the dataset is no longer available. Schroeder et al. studied 65 proprietary automotive Simulink models and found via interviews that engineers preferred simple size metrics such as block count over structural metrics to capture model complexity [148].

5.3.4 SC: First Corpus of Open-Source Simulink Models

Via a two-stage process Chowdhury et al. created what we call SC, the first corpus of freely available Simulink models [22, 31]. First [22], the research team col-

lected 391 models, i.e., 41 of the MathWorks's tutorial models the team considered to not be "toy" examples, the open-source models from MATLAB Central that were most popular (by ratings or downloads), GitHub keyword search results, and 28 models from academic papers, colleagues, and Google searches. Second, the team added the Simulink models of 12 SourceForge repositories and of the 96 most-downloaded MATLAB Central projects, yielding a study of a total of 1,071 Simulink models [31].

SC classifies its 1,071 models as tutorial (41), simple (442), advanced (452), and other (136). The distinction between simple and advanced is determined heuristically: any GitHub project with forks or stars and any MATLAB Central project that are not academic assignment are labeled "Advanced". Models shipped with MATLAB/Simulink are labeled "Tutorial", while models from other sources are labeled 'Other'.

Overall, SC collects Simulink models of projects that (at least partially) are selected and labeled manually. While initially "only" providing project URLs [22], the full corpus [31] includes Simulink model files, metadata, and collection tools and is stored on a Google Drive directory linked from the project's GitHub homepage.

Analyzing the corpus with Simulink R2017a, the work found good modeling practices such as model referencing were not widely used. The work found MathWorks's cyclomatic complexity to be at most moderately correlated[1] with various other model metrics. The correlation was strongest (0.55) for the model's maximum hierarchy depth, followed by the model's number of contained subsystems (NCS). This contrasted with an earlier study by Olszewska et al. [113], which showed strong (0.73)

[1]The earlier work discussed in this paragraph and our own analysis all use Kendall's τ at a 0.05 significance level and follow a recent labeling of subsequent $|\tau|$ ranges at that level, i.e.: "weak" below 0.4, then "moderate" to below 0.7, "strong" to below 0.9, etc. [149]

correlation between MathWorks's cyclomatic complexity and the model's number of contained subsystem (NCS).

5.3.5 SC_{20}: SC Projects Recollected in 2020

In August 2020—some three years after SC was published [31], Boll et al. (a research team acting independently of Chowdhury et al.) collected what we call SC_{20} [89], i.e., the latest Simulink model versions of SC's Simulink projects, yielding 1,734 Simulink models. Simulink models, metadata, and the team's collection tools are preserved on Figshare [150].

The work evaluated SC_{20}'s suitability for empirical model-based research, analyzing each SC_{20} project's domain, origin, and model metrics. The work also proposed a heuristic for identifying models configured for code generation. The paper's analysis found that the majority of SC_{20} models were inadequate for most empirical research, but identified a few mature models. The work also noted that some SC_{20} GitHub projects' characteristics (e.g., a high number of commits and collaborators) suggest potential for evolution research.

5.3.6 SLNET: Largest Known Simulink Corpus

In February 2020 Shrestha et al. collected the SLNET corpus [32], which addresses key issues of SC and SC_{20} (i.e., manual project selection and unclear project licenses), yielding the first redistributable corpus of open-source Simulink models. Specifically, SLNET collects Simulink projects from the GitHub API and from MATLAB Central's RSS feed and does not include projects without Simulink model files, known model generators and their synthetic models, projects that do not have an appropriate license, potentially duplicate projects (via bijection of the projects' models' metrics), and projects whose models all have zero blocks, yielding 9,117 Simulink

models. Simulink models, metadata, and the team's collection tools are preserved on Zenodo [2, 91, 92].

Combining models from the two largest collections of open-source Simulink models (GitHub and MATLAB Central), SLNET is 8 times larger than the largest previous corpus of Simulink models (SC). In March 2023 we confirmed that other hosting sites (still) contain significantly fewer public Simulink repositories (i.e., we could only find 52 Simulink projects on SourceForge and one on GitLab).

5.4 Research Design

Our goal is to gain a deeper understanding of the reproducibility and replicability in model-based development research, particularly regarding Simulink models, as emphasized in a recent literature review [33]. The literature review identified a single study that conducted a large-scale empirical investigation, emphasizing open science, i.e., SC [31]. Subsequently, members of the literature review team undertook their own investigation, by collecting the latest version of the models of the same corpus, i.e., SC_{20} [89].

The recently released SLNET corpus [32] has rectified limitations of the two existing corpora, allowing us to replicate the results of earlier empirical studies. Thus, we perform a sample study utilizing the existing corpora and employ a statistical learning strategy to generalize the findings of prior studies on a smaller dataset to a larger dataset [151,152]. As such, our replication efforts serve a confirmatory purpose.

To structure our study effectively, we have formulated two primary research questions that center around reproducibility and replication.

I What challenges and implications arise when attempting to reproduce model-based development research, specifically for Simulink models?

II To what extent can we generalize prior studies' findings to a dataset that is open-source or larger?

RQ1 In terms of basic Simulink model metrics, how does SLNET compare with earlier open-source corpora and what we know about industrial models?

RQ2 Is SLNET suitable for empirical studies of Simulink projects and their change histories?

RQ3 How do empirical results obtained on smaller open-source corpora and closed-source industry models carry over to the larger SLNET corpus?

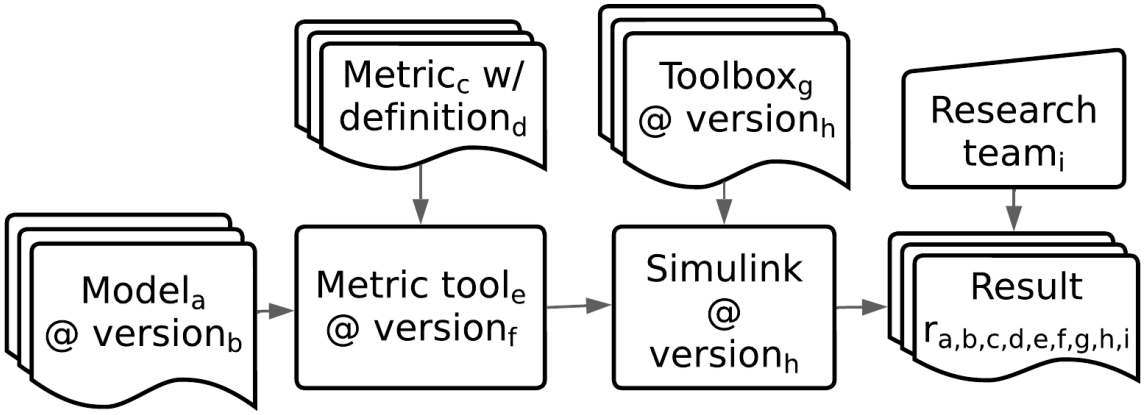


Figure 5.2: Parameters a through i for reproducing and replicating results on Simulink models. Relative to earlier studies (and unless noted otherwise), for reproduction we only varied i and for replication we only varied a,b,i.

Figure 5.2 applies ACM's guidelines on reproducibility ("different team, same experimental setup") and replicability ("different team, different experimental setup") to empirical studies of Simulink models and summarizes the relevant variables. The following sections point out where we had to deviate from this model (e.g., when an exact earlier corpus is no longer available for exact reproduction).

Table 5.1: Overview of three existing (top) plus our four new or re-collected corpora (bottom) of open-source Simulink models; cut-off = date of latest model version in corpus; × = cannot distribute due to unclear licenses.

Corpus	Version of Simulink Models	Cut-off	Data
SC	Original corpus	2017	[153]
SLNET	Larger corpus	Feb '20	[2]
SC_{20}	SC re-collected at later version	Aug '20	[150]
SC_{R}	SC re-collected at SC's version	2017	×
SC_{20R}	SC_{20} completed at SC_{20}'s version	Aug '20	×
$SC_{20REvol}$	SC_{20R} GitHub projects' Git histories	Apr '23	×
$SLNET_{Evol}$	SLNET GitHub projects' Git histories	Apr '23	[125]

5.5 Corpora to Reproduce & Replicate Results

Table 5.1 summarizes the corpora of this study. Boll et al. [89] highlighted that the SC study results had several inconsistencies and Shrestha et al. [32] claimed earlier corpora suffer from unintended human errors and bias. Since both claims lacked sufficient empirical evidence, we attempted to reproduce these studies.

5.5.1 SC_{R} Corpus to Reproduce SC Results

To reproduce the SC study results, we downloaded all models and metadata from SC's Google Drive [153], yielding 1,347 models. This did not include all of the original study's 1,071 models, as the SC distribution excludes 169 models for their unclear licenses. We use SC's source metadata (for 862 of 1,071 models SC lists project URL and version, models studied within the project, and MATLAB version requirements) and retrieve 142 of 169 of these unclearly-licensed models from GitHub (at the same version as in SC).

For 40 of 1,071 models the download included multiple model versions but the metadata did not specify which version was used in the SC study. Since SC only provides aggregated model metrics (instead of per-model measurements), we

could not disambiguate same-name models via comparing the metrics. After receiving confirmation from the SC team, we add all 113 potential model name matches from the SC download, yielding 1,117 models in SC_R (but still missing 27 of 1,071 now inaccessible GitHub models).

Due to the above model name ambiguity (or human error in SC creation), 5 of 1,071 models are now categorized as both Simple and Advanced. Since the SC study reported results per model category, we focused our reproduction on the one category not affected by the above missing/duplicate model issues, i.e., the 41 models labeled "tutorial". Since these 41 models ship with Simulink and we have access to earlier Simulink releases, it was straight-forward to reproduce the SC study results on the same version of the same models on the same Simulink version as the SC study.

The SC paper states that some reported metric results come from a third-party tool [101]. But we found the tutorial models' reported metrics instead exactly match the results of only running the SC metrics tool (which calls the Simulink API [154]). Specifically, we ran the SLNET-Metrics tool [92] as it can run SC's metric tool in the Simulink toolbox configuration [155] the SC study used, yielding the reported 10,926 blocks (as opposed to 10,391 the other tool returns [101]). After this calibration on the tutorial models we ran the SC tool in the same configuration on the rest of SC_R.

Finally, we clarified with the SC team SC's "S-function reuse rate", which SC defined to approximate how often a model contains an S-function it contains elsewhere. The metric basically counts how many S-function blocks in a model have the same name. For example, if a model contains four S-function blocks, three named "a" and one named "b", the reuse rate would be (3-1 + 1-1)/(3+1) = 0.5. SC reported a median reuse rate below 0.5%. Our result on SC_R being much higher triggered an interaction, in which the SC team confirmed that the SC paper mistakenly added the percentage symbol.

5.5.2 SC_{20R} & $SC_{20REvol}$ Corpora to Reproduce SC_{20} Results

We obtained the SC_{20} replication package (v2) from Figshare [150], which contains 1,736 models grouped into 194 projects with non-model files removed. The SC_{20} team categorized projects into four groups based on affiliation: 112 academic, 34 industry-mathworks, 25 industry, and 23 no-information. We included one project with an unknown category in the 'no-information' category, yielding SC_{20R}.

To extract model metrics, SC and SC_{20} mostly use the Simulink API, but there are differences. For instance, SC_{20} counts blocks via sldiagnostics [156] while SC uses Simulink Check [154] (the counts can differ). Additionally, SC uses the Simulink API for cyclomatic complexity, while SC_{20} implements McCabe's definition (independent paths). From the SC_{20} paper [89] and our correspondence with the SC_{20} team we could not reconstruct how SC_{20} computed project-level cyclomatic complexity.

The remaining model metrics we reproduced using the provided tool and documentation. To run the tool we had to install Simulink R2020a and the Check toolbox. We observed discrepancies in the results of 11/1736 models, which we attribute to a lack of documentation regarding the exact Simulink configuration (i.e., toolbox, library, etc).

The SC_{20} team analyzed 35 GitHub projects, but didn't include the necessary git repositories or commit extraction tool in the replication package. We independently developed the tool, and after contacting the authors, they updated their package, but the repositories remained missing. In April 2023 via metadata we obtained 32/35 repositories ("$SC_{20REvol}$"). 3/35 repositories were no longer online.

Finding 1: *The SC and SC_{20} replication packages are insufficient to reproduce the original studies' results.*

Implication: *Authors should host the replication package in permanent archival repositories for long-term access and preservation with documentation, such as Simulink configuration and instruction [157].*

5.5.3 SLNET Results & $\text{SLNET}_{\text{Evol}}$ Corpus

While the SLNET paper does not present any specific study or analysis, it does offer a valuable resource in the form of a corpus of Simulink models along with associated metadata on their metrics. In our attempt to reproduce SLNET's metrics, we first downloaded their corpus from Zenodo [2], which consists of 225 GitHub and 2,612 MATLAB Central projects, as well as a SQLite database of metadata. Following their documentation on Simulink configuration [100], we ran SLNET-Metrics, SLNET's metric collection tool, first on R2018b and then R2019b, as the latter ignores 'resource' folder, which some older SLNET projects use. By following this process, we were able to reproduce their reported metrics.

Like SC_{20}, SLNET only offers project snapshots, but to assess its suitability for evolution studies, we require its git repositories. In April 2023 we obtained 208/225 SLNET GitHub repositories, as 17 projects were offline. We refer to this collection as $\text{SLNET}_{\text{Evol}}$, which we have made available for other researchers to analyze.

5.5.4 Issues in Simulink Tool-chain Found

While trying to reproduce SLNET's results, we encountered the following two Simulink issues. MathWorks classified the first one as a bug and the second one as a documentation issue. First, when using multiple machines to speed up metric collection, Simulink R2018b crashed while compiling a SLNET model on Windows but

compiled the model without issue on Ubuntu. We reported this issue (#04254318), which MathWorks confirmed as a bug and fixed in Simulink R2021b.

Finally, we reported (#04386513) that the cyclomatic complexity definition of the multiport switch [143] did not seem to match Simulink's metrics results. MathWorks addressed this issue by updating its public metric description [144].

5.6 Replicating Empirical Results Using SLNET

To date, empirical data on Simulink models and projects have been obtained on select closed-source projects and smaller open-source corpora (i.e., SC and SC_{20}). We would thus like to know how these earlier results generalize to the larger SLNET corpus of 2,837 open-source projects and their 9,117 Simulink models. As earlier work has not characterized SLNET, we will first put it into context for any subsequent findings or comparisons.

As in similar comparative studies, when interpreting experimental results we need to know how much results are skewed by differences in experimental setups. While conceptually straight-forward, calculating Simulink metrics is influenced by many parameters (Figure 5.2) and we realized that earlier studies did not document all relevant parameter values.

To increase confidence in our results we replicate earlier experiments where possible. Unless noted differently we apply the same metric extraction setup to all corpora—i.e., the same of our researchers use a single consistent set of metric definitions, metric tool version (SLNET-Metrics), Simulink version (R2020b on Ubuntu 18.04), and toolboxes [158].

We used Simulink R2020b as it enhanced metric calculation [159]. For example, in Simulink R2019b a video surveillance system's [160] cyclomatic complexity is 38,403, which on manual inspection seems highly inflated. For the same sys-

tem Simulink R2020b returns 322. Such a drastic change makes it hard to directly compare our results with results reported elsewhere, e.g., the SLNET work used Simulink R2019b [32].

> ***Finding 2:*** *Small changes in experimental setup can drastically skew Simulink model metrics. In one example, upgrading to a newer version of Simulink changed a model's cyclomatic complexity from 38,403 to 322.*
> ***Implication:*** *There are subtle but severe pitfalls when comparing Simulink metric results across papers. To increase confidence in such comparisons we thus repeat earlier experiments where possible.*

5.6.1 Removing User-defined Libraries And Test Harnesses

User-defined libraries and test harnesses serve different goals than regular Simulink models. As they are also structurally different, we first identify and separate them from the regular models. While user-defined libraries are interesting themselves, for analyzing regular models we treat user-defined libraries like all other libraries. We thus either inline blocks from all or none of the libraries. Following prior work [31], we use the Simulink API [161] and identify 235 user-defined libraries in SC_R, 411 in SC_{20R}, and 1,137 in SLNET.

Simulink's Test API [162] can identify models as test harnesses and we thus remove 9 test harnesses from SLNET and two each from SC_R and SC_{20R}. This is likely an under-count, as many open-source projects may not have the license necessary for this API and thus use workarounds. We thus heuristically label (but not remove) models as potential test harnesses by checking if model and folder names contain "test" or "harness", thereby labeling 143 models in SC_R, 233 in SC_{20R}, and 903 in SLNET.

5.6.2 RQ1: Basic Simulink Model Metrics of Corpora

At a high level, while it contains significantly more models, SLNET is not a superset of the previous open-source corpora. Even when containing the same model, corpora may differ in the included model version, due to different corpus collection times. When treating all versions of a model as the same model and including user-defined library models, SLNET contains 30% of the SC models (328/1071), 36% for SC_R (402/1117), 28% for SC_{20} (492/1734), and 28% for SC_{20R} (492/1736).

Table 5.2: Model metrics after removing library & test harness models in SC_R (top), SC_{20R} (middle), and SLNET (bottom); M = models; Mc = models compiling in our setup; Mh = hierarchical models; C = non-hidden connections; $\llcorner^{t0}$ = via SC's metric tool; var = variable; nor = normal; ext = external; PIL = processor in the loop; ac = accelerator; rap = rapid accelerator; Industry-M = Industry Mathworks; M-Central = MATLAB Central; excludes 14 SLNET models that crash Simulink R2020b; includes 20 SLNET models for which Simulink R2020b does not show solver and simulation metrics.

	Models		Hierarchical		Blocks		Connections		Solver Step		Simulation Mode				
	M	Mc	Mh	Mh^{t0}	B	B^{t0}	C	C^{t0}	fixed	var	nor	ext	PIL	ac	rap
Tutorial	41	41	37	40	3,703	13,917	3,700	14,020	13	28	41	0	0	0	0
GitHub	165	92	53	151	7,350	20,734	7,967	21,500	60	105	162	2	0	1	0
M-Central	674	294	488	595	76,473	483,645	80,683	473,466	257	417	655	14	1	4	0
SourceForge	230	33	196	201	18,444	126,123	17,800	125,021	183	47	175	55	0	0	0
Other	7	4	3	7	611	680	636	701	1	6	7	0	0	0	0
$\sum$ SC_R	1,117	464	777	994	106,581	645,099	110,786	634,708	514	603	1,040	71	1	5	0
Academic	690	232	456	634	75,813	185,574	86,223	185,733	229	461	597	68	0	16	9
Industry-M	404	61	259	351	30,826	220,011	27,631	212,299	176	228	399	4	1	0	0
Industry	174	15	93	161	24,753	180,929	25,116	194,655	135	39	169	3	0	1	1
No info	55	24	44	46	4,889	26,690	5,524	26,803	29	26	54	1	0	0	0
$\sum$ SC_{20R}	1,323	332	852	1,192	136,281	613,204	144,494	619,490	569	754	1,219	76	1	17	10
GitHub	1,637	541	875	1,297	190,213	424,175	188,069	400,753	860	759	1,498	103	2	14	2
M-Central	6,239	3,370	3,874	5,485	828,210	3,197,090	914,857	3,074,782	1,753	4,484	5,971	186	2	76	2
$\sum$ SLNET	7,876	3,911	4,749	6,782	1,018,423	3,621,265	1,102,926	3,475,535	2,613	5,243	7,469	289	4	90	4

The remainder of this work removes from each corpus each model that is a test harness or a user-defined library. This differs from earlier work that treated user-defined library models as regular models and thus included them in overall metric counts [89]. (The only exceptions are the three Table 5.2 $\llcorner^{t0}$ columns, which in-

line user-defined libraries.) Table 5.2 compares SC_R, SC_{20R}, and SLNET on basic Simulink model metrics, such as number of models, models that are hierarchical, blocks, connections, and solver and simulation modes.

5.6.2.1 Model Size

A widely-used proxy for model size is the model's number of blocks [163–165]. For example, a recent paper conducted experiments on what it introduced as large industrial automotive models, containing 3.7k–73k blocks (and having hierarchy depth 8–16) [166]. Boll et al. report conversations with Simulink experts indicating typical industrial models often have 1k–10k blocks [33]. Industry-scale models at automotive supplier Delphi were earlier reported to have on average some 750 blocks [167].

Table 5.2 shows that (except for "Others"), including imported library blocks (B^{t0}) at least doubles the overall block count. Focusing on 1k+ block models, SC's custom tool (which includes imported library blocks) found 93 such models in SC on Simulink R2017a. On Simulink 2020b, SC's tool found 132 such models in SC_R, 139 in SC_{20R}, and 799 in SLNET. When excluding any imported library blocks, SC_R contains 14 such models, SC_{20R} 15, and SLNET 148.

5.6.2.2 Hierarchical & Compiling Models

Model hierarchy is important for studying model complexity, model slicing and evaluating Simulink model generation tools [29,60,109,168]. SC_R has 777 hierarchical models, of which we could compile 44%. Of SC_{20R}'s 852 hierarchical models we could only compile 20%. Of SLNET's 4.7k hierarchical models we could compile 47%. SC_{20R}'s low compile rate can be attributed to that corpus not distributing non-model files, which may have served as dependencies for the Simulink model.

5.6.2.3 Project and Model Metric Distributions

Table 5.3: Model (after removing library & test harness models) metric distributions per project (p) and per model (m) in SC_R (R), SC_{20R} (20R), and SLNET (N); Cyclom. C. = cyclomatic complexity (for a project the max of its models); Model Ref. = model references; Alg. L. = algebraic loops; LL Blocks = library linked blocks; Sub. Blocks = blocks in a subsystem at depth that has most such blocks.

		Min			Max			Average			Median			Standard Deviation		
		R	20R	N	R	20R	N	R	20R	N	R	20R	N	R	20R	N
Models	p	1	1	1	124	124	237	5.6	6.9	2.8	1.0	1.0	1.0	14.7	16.4	9.7
Blocks	p	1	1	0	13,555	13,831	172,196	457.4	706.1	362.8	116.0	140.0	52.0	1,419.9	1,959.8	3,577.1
	m	1	0	0	13,555	13,555	18,255	95.4	103.0	129.3	25.0	25.0	27.0	448.4	430.6	690.1
Block types	p	1	1	1	55	58	104	18.3	19.2	13.2	16.0	17.0	11.0	11.1	11.9	9.3
	m	1	1	1	47	47	101	10.6	10.4	10.4	8.0	8.0	8.0	8.3	7.8	7.9
Connections	p	0	0	0	14,169	16,491	231,672	475.5	748.7	392.9	124.0	153.0	57.0	1,422.1	2,103.5	4,611.7
	m	0	0	0	14,169	14,169	25,078	99.2	109.2	140.0	26.0	27.0	28.0	466.2	453.7	887.1
Subsystems	p	0	0	0	1,809	1,873	19,622	46.9	68.4	34.0	7.0	7.0	2.0	179.3	210.8	414.1
	m	0	0	0	1,294	1,294	2,117	9.8	10.0	12.1	3.0	2.0	2.0	44.1	41.8	75.3
Cyclom. C.	p	0	0	0	322	322	2,404	27.7	30.7	22.2	7.0	7.0	5.0	49.4	54.1	81.1
	m	0	0	0	322	322	2,404	14.0	13.6	13.7	4.0	4.5	2.0	32.4	31.6	59.0
Model Ref.	p	0	0	0	4	10	54	0.1	0.1	0.1	0.0	0.0	0.0	0.4	0.8	1.5
	m	0	0	0	4	2	12	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.1	0.4
Alg. L.	p	0	0	0	7	9	37	0.2	0.2	0.1	0.0	0.0	0.0	0.7	1.0	1.1
	m	0	0	0	2	1	6	0.1	0.1	0.1	0.0	0.0	0.0	0.2	0.2	0.3
LL Blocks	p	0	0	0	657	423	2,311	9.1	11.8	5.6	0.0	0.0	0.0	53.8	48.7	81.1
	m	0	0	0	31	31	441	1.9	1.7	2.0	0.0	0.0	0.0	4.5	4.3	15.0
Sub. Blocks	-	2	2	3	21	21	100	9.6	9.5	9.1	11.0	11.0	7.0	3.9	3.9	11.5

Table 5.3 shows model metric distributions across SC_R, SC_{20R}, and SLNET. The majority of SLNET models are relatively small, with mean exceeding median values. The overall distribution of metrics in SLNET is akin to that of earlier corpora, i.e., offering a broad spectrum with most standard deviations exceeding the means. SLNET however offers a broader range of Simulink models with similar min but notably larger max metric values. Following are additional distribution details of project size, most frequently used block types, and file types.

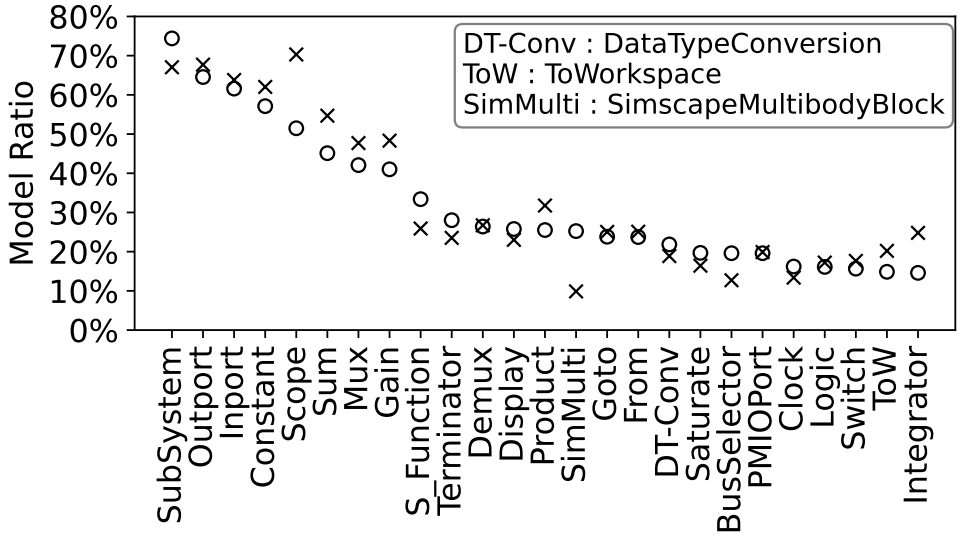
5.6.2.3.1 Project size Similar to earlier corpora, the distribution of models in SLNET is skewed towards a few large projects. The 50 largest projects (i.e., the

largest 1.8% of projects) contain 35% of all models, while 76% of the projects contain just one model. Some SLNET projects feature 18 empty models alongside non-empty models. By comparison, in SC_{20R}, 5/194 projects contain 35% of the models, and 53% of the projects contain just one model. With the exception of a single SLNET project that comprises a library model, all projects include some blocks and signal lines.

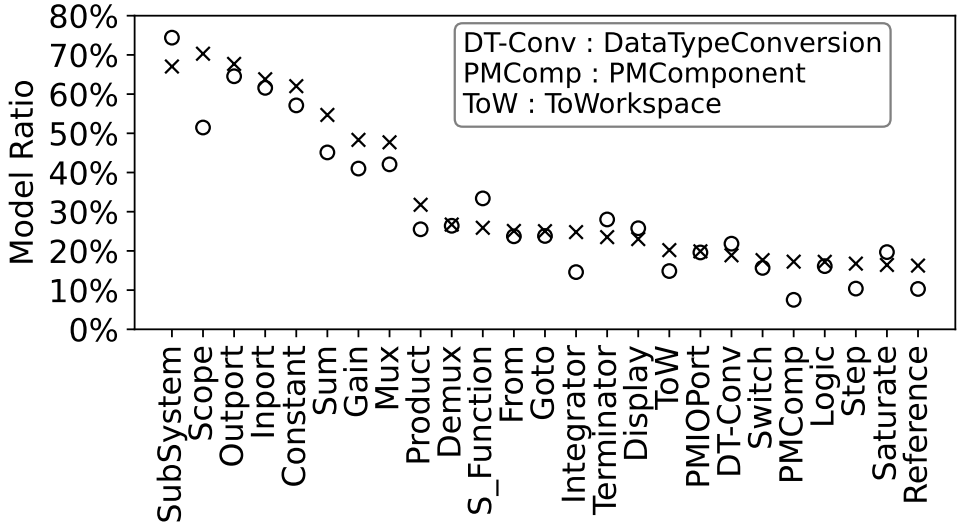
5.6.2.3.2 Most Frequently Used Block Types Figures 5.3a and 5.3b show that the distributions of the most-commonly used block types are similar in SC_R and SLNET. For example, in each corpus over 60% of models contain a SubSystem block, making SubSystem appear in the most models in both corpora. SLNET uses SubSystem less-widely, likely as 28% of SLNET models have less than 8 blocks, which typically does not require a SubSystem block.

SC_R models use 156 distinct block types vs. 203 in SLNET (150 are in both). SLNET thus offers a potentially valuable resource for research studies [22, 169]. Both SC and SC_{20} studies included library-imported blocks and reported a lower occurrence of output blocks (e.g., Scope [170], Display [171], and ToWorkspace [172]) than SLNET. The possible explanation for this discrepancy is that, like in procedural programming languages (where programmers include logging statements at various execution points), libraries may not have such statements for efficiency purposes. This practice is also observed in Simulink modeling.

Furthermore, From [173] and Goto [174] blocks, which are typically used to improve the visual layout of the model, are equally widely used in SC and SLNET. However, excessive non-local usage of From and Goto blocks adversely affects readability and design, warranting further investigation.



(a) Most-common block types in SC_R (o) and their SLNET (x) rate.



(b) Most-common block types in SLNET (x) and their SC_R (o) rate.

Figure 5.3: Most-common block types in SC_R (a) and SLNET (b).

5.6.2.3.3 File types Each Simulink model is stored in one of two file formats, the MDL legacy file format or SLX. Introduced in Simulink R2012a, SLX conforms to the Open Packaging Conventions (OPC) interoperability standard. Across corpora, few projects contain both MDL and SLX files (SC_R 3%, SC_{20R} 7%, and SLNET 2%).

Overall the major file type has shifted from MDL in SC_R to SLX in SLNET (39% of SC_R models are in SLX, 45% for SC_{20R}, and 55% for SLNET). The prevalence of SLX files in open-source models is significant for developing SLX to MDL back-transformation tools [175].

In summary, SLNET shares many similarities with prior corpora and offers a broader view of open-source Simulink projects. The majority of SLNET models are small, which may be relevant for analyzing simple models [28, 29, 176, 177], while also including a substantial number of non-trivial models using diverse features.

Finding 3: *As in many other kinds of open-source projects [9, 178], SLNET project and model metrics follow long-tailed distributions.*
Implication: *Research studies may use SLNET subsets based on their objectives. The diverse SLNET corpus can help address generalizability challenges in model-based development research.*

5.7 Replicating Findings on Modeling Practices

5.7.1 Converging Result: Model Referencing

Analogous to classes in object-oriented programming, model references [179] enable modular model design, unit testing, and code reuse. But similar to the SC work [31], we found that only 10 SC_R (0.9%), 18 SC_{20R} (1.4%), and 139 SLNET models (1.8%) use model referencing. Even when accounting for the skewed SLNET model size distribution, Table 5.3 shows that model reference use remains sparse.

5.7.2 Converging Result: Algebraic Loops

An algebraic loop arises from a circular dependency between a block's output and input at the same simulation time step. An algebraic loop may reduce simulation

performance or prevent the solver from resolving the loop. As the SC work [31], we found such loops relatively rarely, with only 20 SC_R and 186 SLNET models containing such loops.

5.7.3 Converging Result: Small Class Phenomenon

Zhang et al. observed the "small class" phenomenon in Java programs (most classes have few lines of code while a few classes are large) and found a high correlation between class size and number of defects [180, 181]. In Simulink, subsystems are used to encapsulate a function, resulting in a hierarchical model. Similar to the small class phenomenon noted in the SC work [31], we observe that the median number of blocks in a subsystem at any hierarchy does not exceed 11 in both SC_R and SLNET. This may inform future hypotheses on Simulink subsystem size and defects.

> ***Finding 4:*** *The median number of blocks in a subsystem at any hierarchy level does not exceed 11.*
>
> ***Implication:*** *More research is needed to assess how subsystem size impacts Simulink model quality.*

5.7.4 Converging Result: S-function Reuse Rate

Table 5.4: S-function per-model reuse rate for models with 1+ S-functions; $M_{S\text{-}fct}$ = models with 1+ S-functions; LQ = lower quartile; UQ = upper quartile; med = median.

	$M_{S\text{-}fct}$	min	LQ	med	UQ	max	avg
SC_R	351	0.0	0.0	0.0	0.38	0.92	0.20
SC_{20R}	378	0.0	0.0	0.0	0.50	0.98	0.23
SLNET	1,504	0.0	0.0	0.0	0.50	0.99	0.21

Besides reuse of legacy C code, S-functions allow within-model code reuse (i.e., defined once but added to and used in several model components). In the same spirit as the SC work [31], Table 5.4 shows that S-functions are not widely used, with just 31% of SC_R models and 20% of SLNET using S-functions. For models that use S-functions, 41% of SC_R models and 40% of SLNET models reuse at least one S-function (but these models' median S-function reuse rate is zero across corpora).

5.7.5 Diverging Result: Cyclomatic Complexity vs Other Metrics

We conduct a correlation analysis between cyclomatic complexity and the other Table 5.5 model metrics using Kendall's τ. We only use models for which we could calculate cyclomatic complexity (e.g., excluding models we could not compile). As in the SC study, for SC_R we used non-Simple models. For SC_{20R}, we used industry and industry-MathWorks models. As SLNET models are not categorized, we used those containing 200+ blocks. All metrics exhibit a statistically significant correlation at a 0.05 significance level.

Table 5.5: Correlation between cyclomatic complexity and model metrics; M, B, C from Table 5.2: models, blocks, and non-hidden connections; UB = unique block types; MHD = max. hierarchy depth; CRB = child-model representing blocks i.e., model reference and subsystem; NCS = contained subsystems.

	M	B	C	UB	MHD	CRB	NCS
SC_R	160	0.29	0.32	0.31	**0.38**	0.28	0.29
SC_{20R}	58	0.16	0.16	0.20	0.31	**0.41**	**0.41**
$SLNET_{\geq 200}$	279	0.27	0.27	0.23	0.10	**0.28**	0.27
$SLNET_{200\text{-}300}$	111	-0.02	0.12	**0.16**	0.05	0.07	0.07

SC_R models have a weak positive correlation (0.28 to 0.38) between cyclomatic complexity and model metrics. For SC_{20R} models the correlation is positive and weak

to (barely) moderate (0.16 to 0.41). For SLNET models with 200+ blocks the correlation is positive but remains weak (0.10 to 0.28).

> ***Finding 5:*** *Contrary to previous work [113], cyclomatic complexity does not seem strongly correlated with other model metrics.*
>
> ***Implication:*** *Similar to Java- and C-like languages, in Simulink cyclomatic complexity seems to remain an independently valuable metric.*

Table 5.6: $\text{SLNET}_{\text{Evol}}$ and $\text{SC}_{\text{20REvol}}$ per-model (m) and per-project (p) change metrics; Total commits, commits per day during project duration, merge commits ($\succ$), and commits of 1+ mdl/slx files (MS); commit authors and $\text{commit}_{\text{MS}}$ authors; med = median; std = standard deviation.

		$\text{SC}_{\text{20REvol}}$					$\text{SLNET}_{\text{Evol}}$				
		min	max	avg	med	std	min	max	avg	med	std
Commits	p	1	590	62.7	10.5	124.6	1	963	43.9	7.5	120.1
Commit / day	p	0	4	0.9	0.3	1.2	0	24	1.9	0.6	3.1
$\text{Commits}_{\text{MS}}$ [%]	p	0	100	38.8	26.8	31.2	1	100	31.4	25.0	23.5
$\text{Commits}_{\succ}$ [%]	p	0	17	2.7	0.0	4.7	0	40	3.2	0.0	6.7
$\text{Updates}_{\text{MS}}$	m	0	43	3.3	1.0	5.7	0	53	1.8	1.0	2.8
Authors	p	1	16	2.8	2.0	3.5	1	21	2.0	1.0	2.6
	m	1	3	1.1	1.0	0.4	1	8	1.3	1.0	0.7
$\text{Authors}_{\text{MS}}$ [%]	p	0	100	68.6	75.0	34.5	10	100	82.2	100.0	26.1

5.7.6 Converging Result: Suitability For Change Studies

To assess their applicability for Simulink model and project change studies, we analyzed $\text{SC}_{\text{20REvol}}$'s 32 and $\text{SLNET}_{\text{Evol}}$'s 208 git repositories (for $\text{SLNET}_{\text{Evol}}$ we only studied the commits until SLNET's February 2020 snapshot). Three projects (with 811 commits) were in both corpora.

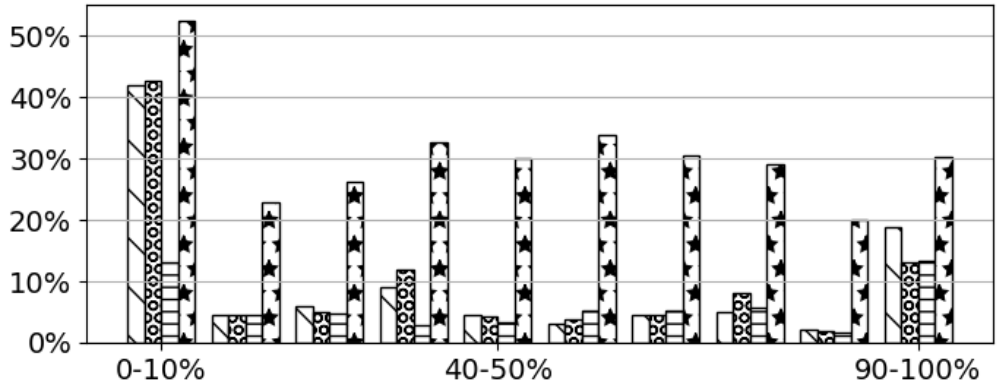
Table 5.6 gives an overview of the project and model change metrics. For example, 53% of $\text{SC}_{\text{20REvol}}$ projects (17/32) and 39% of SLNET projects (82/208)

are maintained by at least two collaborators, of which 8/17 and 32/82 have commits spanning over a year. Just 22% of $SC_{20REvol}$ and 15% of $SLNET_{Evol}$ projects have more than 50 commits. Across $SC_{20REvol}$ and $SLNET_{Evol}$ projects, 20% of commits involved updates or the creation of one or more models.

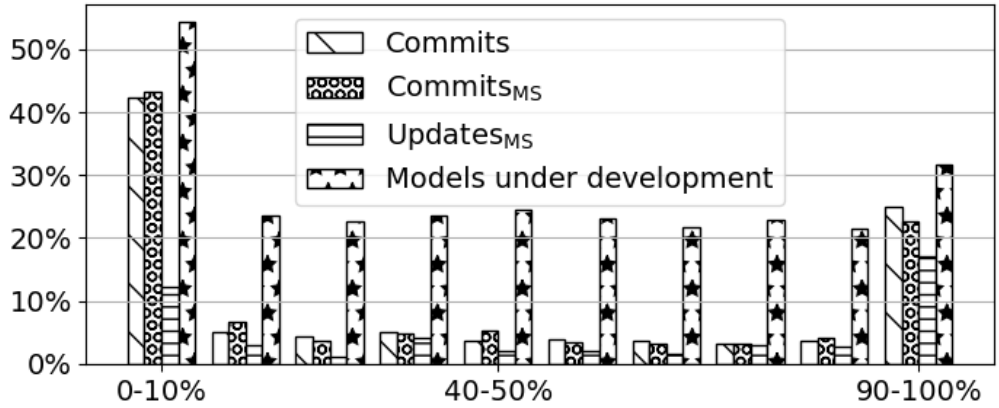
In both corpora, an average of 22% of models were under active development throughout the projects with 3+ commits, indicating the models were primary artifacts of these projects. However, 40% of $SC_{20REvol}$ and almost half of $SLNET_{Evol}$ projects did not update their models after committing them to the repository. In both corpora, roughly 55% of models were not updated at all. The lack of model updates may be due to GitHub Simulink projects mainly serving as archives—like most other GitHub projects [9].

Figure 5.4 breaks each project's duration into 10 buckets of equal length (normalized to each project's duration). Here project duration is the duration from a project's first to last commit as recorded by the timestamps assigned by the authors' machines. While this approach has its pitfalls, the more-active projects are usually less affected and we performed the basic recommended sanity checks to ensure there are no impossible outliers (e.g., commits with Unix time zero) [182].

To avoid potential skewing caused by "code dump" projects, Figure 5.4 excludes projects with less than 3 commits, yielding 26 $SC_{20REvol}$ projects and 186 $SLNET_{Evol}$ projects. Even with this filtering, the figure may still be biased towards projects with fewer commits as the majority of both $SC_{20REvol}$ and $SLNET_{Evol}$ projects have less than 11 commits.



(a) Timeline of 26/32 $SC_{20RExvol}$ 3+ commit projects.



(b) Timeline of 186/208 $SLNET_{Evol}$ 3+ commit projects.

Figure 5.4: Across normalized project duration (x-axis): Total project commits, commits of 1+ mdl/slx files, individual mdl/slx file updates, and mdl/slx files under development (i.e., in between a file's first and last commit).

Finding 6: *A quarter of $SLNET_{Evol}$ projects are developed collaboratively and have 1+ multi-revision models.*

Implication: *$SLNET_{Evol}$ projects have the potential to yield valuable insight into open-source Simulink development.*

5.7.7 Diverging Result: Open-source Code Generation Models

Simulink models that can generate code are of interest in model-based research and tool-development [164, 183–186]. Initially we applied SC_{20}'s heuristics to search

Table 5.7: Models configured for code generation; M = all models; EC_{20} = SC_{20} Embedded Coder heuristics; EC = our Embedded Coder heuristics; GRT = Simulink Coder (Real-Time Workshop); Other = other code generation toolboxes.

	M	EC_{20}	EC	GRT	Other	Total
Tutorial	41	1	1	12	0	13
GitHub	165	0	4	52	4	60
MATC	674	0	47	101	109	257
Sourceforge	230	0	0	96	87	183
Others	7	0	0	1	0	1
$\sum$ SC_R	1,117	1	52	262	200	514
Academic	690	0	3	94	136	233
Industry-M	404	0	33	77	67	177
Industry	174	0	5	129	1	135
No Info	55	0	1	28	0	29
$\sum$ SC_{20R}	1,323	0	42	328	204	574
GitHub	1,637	14	129	502	234	865
MATC	6,239	19	423	1,050	297	1,770
$\sum$ SLNET	7,876	33	552	1,552	531	2,635

for Embedded Coder [187] or TargetLink [188] traces. But we found inconsistencies between SC_{20}'s results (finding no code generation models) and their replication package's heuristics [189]. During our interactions the SC_{20} team acknowledged a bug and fixed it in their replication package version 2 [150].

Specifically, SC_{20}'s heuristics determine if a model can generate code based on the presence of *atomic* subsystems [190] or special TargetLink blocks. This found 33 SLNET models configured for Embedded Coder but no TargetLink traces. We found this heuristic restrictive and not specific to Embedded Coder. Our counter-example model had non-atomic subsystems and successfully generated code via Embedded Coder.

For background, while every Simulink model can generate code using Simulink Coder [191], this requires a fixed-step solver, which conflicts with the default variable-step solver model configuration. Simulink models further rely on a target language compiler (TLC) file [192] to map Simulink blocks and parameters to the target language's constructs.

Simulink offers a set of standard-named TLC files that support various solver types [193]. For example,'rsim.tlc' supports fixed-step and variable-step solvers. To determine if the Simulink model is configured to generate code, we follow a heuristic approach. First, we check if the model's TLC file name matches with one provided by Simulink and the model is configured with appropriate solver type. Second, in cases where the solver type required is ambiguous, we make a conservative assumption that the model must be configured with the fixed-step solver.

Table 5.7 shows the number of models configured for code generation. SLNET has 2,635 models with code generation capabilities, at least 4× more than previous corpora.

> ***Finding 7:*** *SLNET has 4× models configured for code generation (a common configuration in industrial models) than the largest earlier open-source model collection.* ***Implication:*** *Additional investigation is required to determine if the code generation models in SLNET can meet requirements of research studies.*

5.8 Threats to Validity

Internal validity concerns the experimental design, data collection and analysis. In our replication efforts, we closely adhered to the original study's setup and tools. We calibrated the provided tools and contacted the authors for clarification and con-

sistency in data analysis. It is important to note that the choice of Simulink version can impact model metrics and introduce slight differences in insights.

Specifically, for a subset of 554 SLNET models (the models of the 10 SLNET projects with the most models) we compared model metrics obtained using both R2020b and R2022b. Results for all metrics were the same for all models, except for 3/554 models where the cyclomatic complexity differed by 2–6 between R2020b and R2022b.

External validity examines the generalizability of reproduced and replicated study results. In our case, the generalizability of our findings is limited to Simulink models within the SLNET corpus. SLNET may not represent all available Simulink projects, as its construction involved a keyword search on GitHub and filtering for redistributable projects. However, considering that the majority of results from the original studies, which involved some level of cherry picking in their corpus, hold true in SLNET–a larger dataset encompassing diverse models with a small overlap–we are optimistic in the generalizability of the presented results to other open-source Simulink models.

Construct validity ensures that the measures and metrics used in the replicated study accurately capture the intended concepts. Our confirmatory replication study inherits limitations from the original studies, such as not analyzing Stateflow blocks or MATLAB code, which can contribute to the project's complexity. Also, SC's heuristic used to identify test harnesses may have limitations, as manual inspection revealed 10% of such models are test harnesses. Upon noticing issues with SC_{20}'s code generation heuristic, we proposed new methods after consulting with the original authors.

Reliability refers to the replicability of a study for obtaining same or similar results. To mitigate reliability risks, we distribute our analysis tool and complete

replication package as open-source via permanent storage locations [125, 126]. We encourage replication of our findings.

5.9 Conclusions and Future Work

The study investigated the reproducibility of previous empirical studies of Simulink models and evaluated the generalizability of their results to the larger SLNET corpus. The SLNET study confirmed and contradicted earlier findings, highlighting its potential as a valuable corpus for model-based development research and also provided actionable insights for future research. We found that open-source Simulink models generally follow good modeling practices and that few open-source models are comparable in size and properties to proprietary models. To that end, we proposed a heuristic to determine code generating Simulink models. We also provided 208 Git repositories to facilitate model evolution studies.

While this paper only analyzes Simulink model metrics focusing on reproducibility and replication, future work includes examining if the model metrics can be used to make predictions of process metrics such as defect prediction.

CHAPTER 6

EvoSL: A Large Open-Source Corpus of Changes in Simulink Models & Projects

This chapter was originally published in 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS), Västerås, Sweden, 2023, pp. 273-284, doi: 10.1109/MODELS58315.2023.00024. It is reproduced here with permission from IEEE without revision [194].

6.1 Abstract

Having readily available corpora is crucial for performing replication, reproduction, extension, and verification studies of existing research tools and techniques. MATLAB/Simulink is a de-facto standard tool in several safety-critical industries for system modeling and analysis, compiling models to code, and deploying code to embedded hardware. There is no commonly used corpus for large-scale model change studies because there is no readily available corpus. EvoSL is the first large corpus of Simulink projects that includes model and project changes and allows redistribution. EvoSL is available under a permissive open-source license and contains its collection and analysis tools. Using a subset of EvoSL, we replicated a case study of model changes on a single closed-source industrial project.

6.2 Introduction

There is currently no well-packaged, single source of open-source Simulink projects suitable for studying changes in Simulink models or projects. This is primarily due to the overhead associated with mining open-source repositories for such

projects. For instance, GitHub's API does not readily facilitate filtering Simulink projects. Additionally, many open-source projects have been rendered inactive, adding another layer of complexity to the task of filtering out unwanted noise [9]. So creating a centralized and diverse set of Simulink projects is currently challenging.

This is a significant problem, as Simulink is a powerful tool that is widely used in several safety-critical industries such as automotive, aerospace, healthcare, and industrial automation for system modeling and analysis, compiling models to code, and deploying code to embedded hardware. As models become increasingly complex, understanding the impact of changes on the overall system becomes challenging and maintaining the consistency between models becomes harder. To alleviate the problem, researchers often collaborate with industry to study evolution patterns and develop tools and techniques [87, 195, 196]. But this has significantly hampered the advancement of the research as the software artifacts they used are generally not made available due to confidentiality agreements hindering replication, reproduction, extension, and verification of results.

In software engineering research, there has been steady progress towards making research artifacts publicly available, which in turn has increased the impact of the research [197]. The full adoption of the open-source mindset in model-based development research has been limited, due to a prevailing view that publicly accessible models have limited research utility or because of non-disclosure agreements between researchers and industry partners [33]. Recent work has created increasingly larger corpora of open-source Simulink models [22, 31, 85, 198]. A recent empirical study has shed light on the potential of using open-source Simulink models in model evolution studies [89]. However to date these corpora do not contain Simulink model or project change data.

To address the issue, we present EvoSL, a curated corpus of 924 Git repositories consisting of over 140k commits. EvoSL is self-contained and redistributable, automatically (apart from occasional license reviews) collected from GitHub. We demonstrate EvoSL's usefulness by replicating on a EvoSL subset a model evolution study originally performed on an industry project. Our results share several similarities, while also bringing to light significant differences. For instance, our analysis found that engineers spend a substantial amount of time managing signal data rather than implementing algorithms and documentation is often neglected. To summarize, the paper makes the following major contributions.

- We created EvoSL, a corpus of 924 Simulink repositories. We mined GitHub to extract and filter Simulink-based Git repositories that permit redistribution. To the best of our knowledge, EvoSL is the first corpus of third-party Simulink projects to perform model change studies.
- To assess EvoSL's usefulness, we tried to replicate a prior study that analyzed changes of closed-source industrial models. We found several of the original findings could be observed on the open-source models.
- All artifacts of the paper including tools and mined data are open sourced on Zenodo [199, 200] and Figshare [201].

6.3 Background

Simulink [69] is a popular model-based development tool that allows scientists and engineers to design, analyze, and implement complex systems. For example, it is widely used in the aerospace, automotive, healthcare, and robotics industries.

A Simulink user designs a system via a graphical modeling environment as a *block diagram*, by connecting parameterized blocks that represent components, signals, and mathematical operations. Figure 6.1 shows a tiny example. Each block

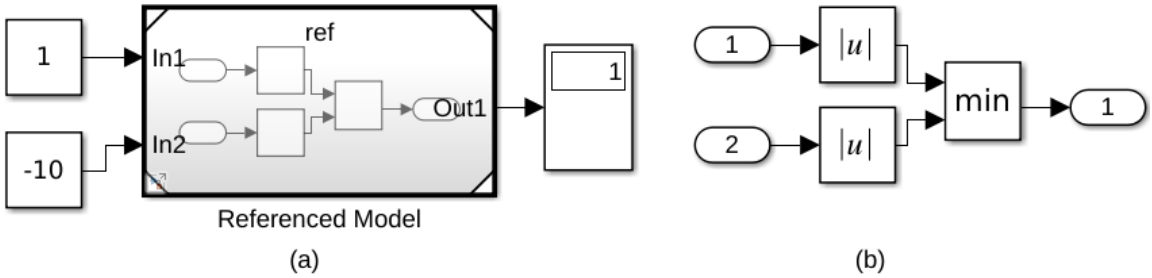


Figure 6.1: Two tiny example Simulink models: The left model (a) reuses the functionality of the referenced model (b).

processes the input it receives via *input ports* and passes its outputs via *output ports* via *signal lines* to subsequent blocks.

Simulink users can pick blocks from a wide variety of libraries and create custom blocks. A block *mask* [202] can add user-defined constraints and user-interface elements to a block. A Simulink user can annotate a model using *annotations* and alter its behavior via *configurations* [203].

Simulink's built-in *Model Comparison Tool* [204] compares two model versions at various levels of granularity—from blocks to the overall model structure. The example in Figure 6.2 shows the differences between two versions of a Simulink tutorial model (sf car [3]), i.e., the addition of an Output block (Out1) and the corresponding update of the Vehicle-to-transmission connection line (partially highlighted in yellow).

6.3.1 Studies of Changes in Simulink Models & Projects

Despite the importance of Simulink in practice, to the best of our knowledge there are only a few studies of how practitioners develop Simulink projects and how Simulink models change during development [4, 89, 205]. While studying the development history of large Simulink projects in both closed-source industrial [4, 205] and open-source development [205], such work has mostly consisted of case studies of one or two projects.

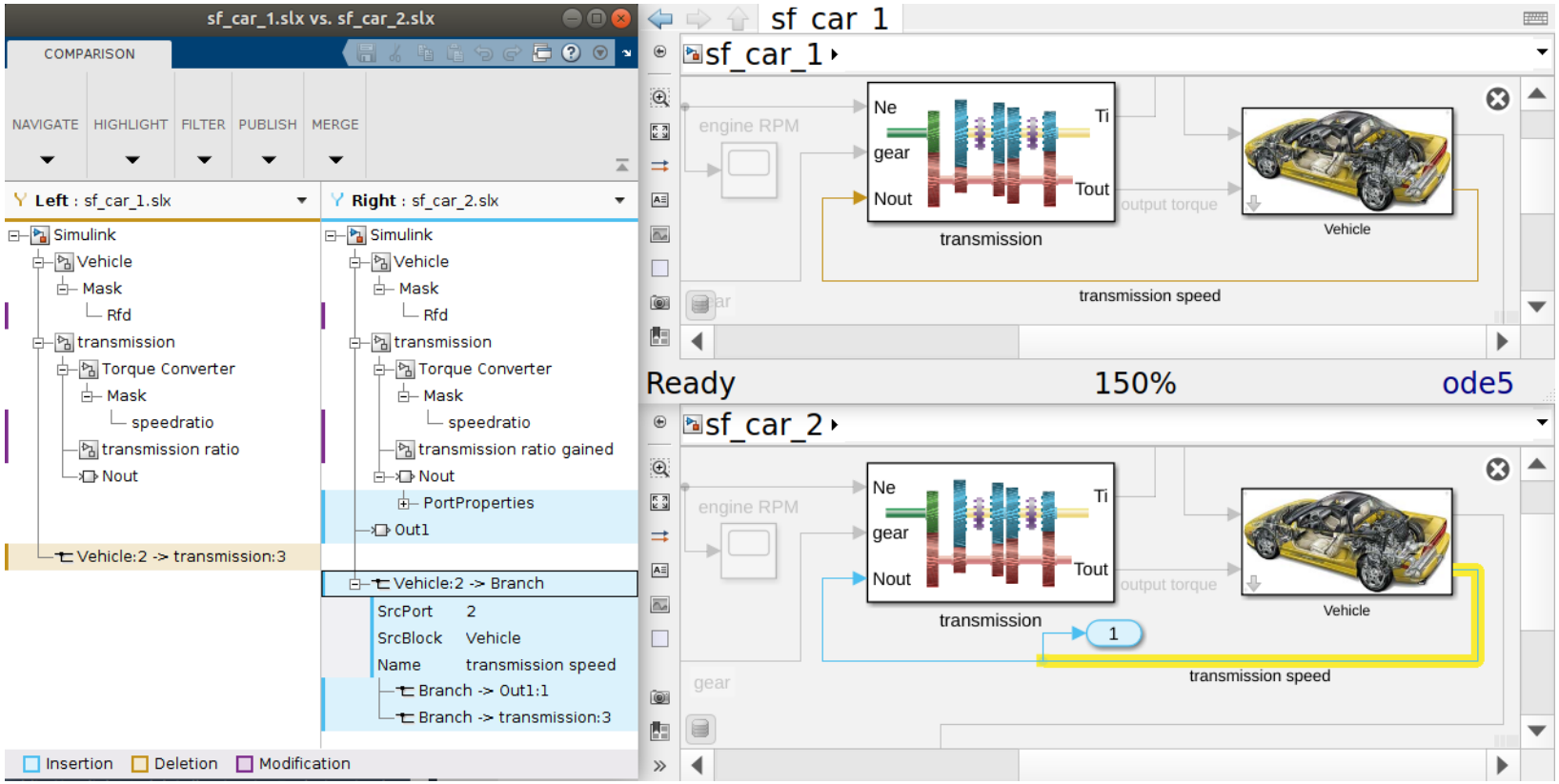


Figure 6.2: Sample Simulink Model Comparison tool report comparing two versions of the sf car Simulink tutorial model [3].

The one exception we are aware of is the recent study [89] that collected the commit histories of 35 open-source Simulink projects of an earlier corpus [31] that were still online. Besides the limited number of projects, the study also remained focused on high-level project history change data and thus did not analyze changes within a model file, e.g., which model elements are changed and how the elements are changed.

6.3.2 State of Open-source Simulink Corpora

While several services provide open-source Simulink models, to the best of our knowledge, none of these services can currently be used directly as a corpus of Simulink project repositories. For example, GitHub does not list Simulink as a separate language, has many projects that are unclearly licensed or are duplicates, and over time many projects disappear. Similarly, Software Heritage is not set up for frequent public download of full repositories and MATLAB Central File Exchange does not support commit-level project histories.

The lack of a corpus of Simulink repositories has been partially addressed by recent efforts to create ever-larger corpora of open-source Simulink models [22, 31, 85, 89]. On the positive side, these corpora have been used as a training set for machine-learning based approaches [176, 198] and to evaluate a variety of novel techniques [60, 198, 206–208]. Unfortunately, these corpora do not contain model changes.

To gauge the promise of open-source projects for studying Simulink project and model changes, Boll et al. [89] studied the Git repositories of 35 GitHub projects of Chowdhury et al.'s corpus [31]. With a Simulink expert many of these 35 projects were found to not mirror industrial Simulink projects for various reasons (i.e., under 50 day project duration, single author, and few merge commits). On the positive side,

the study mentions three projects as promising for—due to their low total number—case study research.

While we could add model changes to one of the above corpora (e.g., by adding Git commits from GitHub), the maximum size of such a complemented corpus would still be relatively limited. Specifically, the corpus with the by-far most potentially available project histories is SLNET with its 225 GitHub projects [85].

6.3.3 Available Open-source Simulink Project Histories

As of March 2023, the major non-GitHub services we are aware of hosting code repositories, GitLab and SourceForge, host orders of magnitude fewer open-source Simulink repositories than GitHub. Specifically, before removing projects that are empty, forks, duplicate, or have an unclear license, a quick search for "Simulink" yields 52 SourceForge projects and one GitLab project.

While Software Heritage preserves many important open-source code repositories long-term, we do not use it for the following reasons. First, Software Heritage is not meant as a primary source, downloading Software Heritage repositories is expensive and should only be done if the primary source becomes unavailable [209]. Second, it contains fewer repositories (i.e., missing over a third of the 14k EvoSL$^+$ Simulink root repositories we located on GitHub). Finally, Software Heritage currently does not provide GitHub project data such as issues, comments, and pull requests.

Beyond centralized project hosting services, a recent study [210] found the three most-used decentralized code repository sources GitLab Community Edition, Gogs, and Gitea to provide over 45k public open-source Git repositories, which tend to be longer-running, more academic, and more collaborative than GitHub projects. We did not attempt to mine these services as overall they had three orders of magnitude

fewer projects than GitHub and do not provide uniform project data such as issues, comments, and pull requests.

6.4 Corpus of Simulink Model & Project Changes

Downloading Simulink projects from GitHub is not straightforward, as GitHub does not label Simulink projects. To heuristically address this issue, EvoSL-Miner queried GitHub's public REST API (via PyGitHub [97]) in February and March 2023 to first download *root* repositories (i.e., not forks) that (a) are marked as using the MATLAB programming language or (b) match when searching their repository name, description, or README file for "Simulink". EvoSL-Miner extends SLNET-Miner, by downloading the full Git repository instead of a snapshot, while still satisfying the GitHub REST API limits (30 search requests per minute per authenticated user, yielding 1k results per request; 5k other requests per hour per authenticated user) [211]. This yielded over 360k such MATLAB/"Simulink" projects.

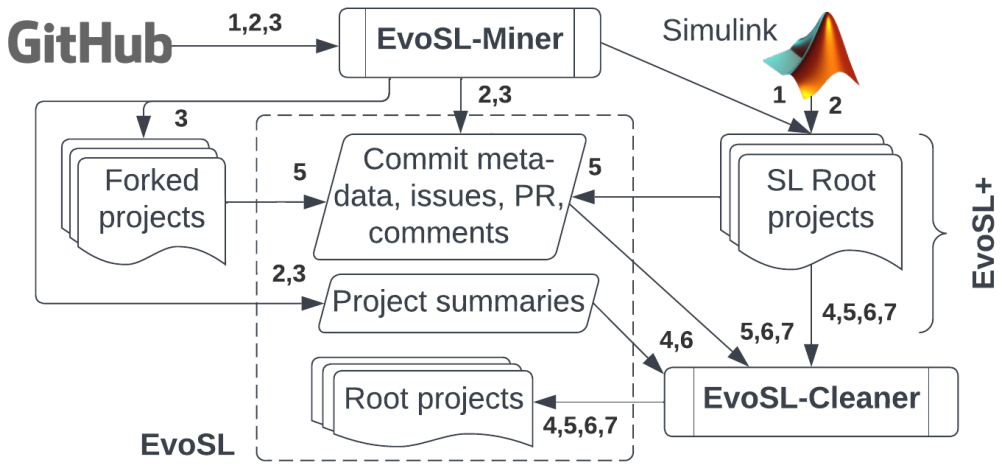


Figure 6.3: Overview of EvoSL collection and cleaning steps: EvoSL-Miner downloads EvoSL$^+$ (Git projects and metadata), from which EvoSL-Cleaner removes certain Git repositories.

In the second step (labeled "2" in Figure 6.3), we only keep a MATLAB / "Simulink" project in EvoSL$^+$ (and download from the GitHub API its summary data, issues, pull requests, and comments) if the project's latest default-branch version has at least one mdl or slx file we can open with Simulink 2022b (Simulink's default file format changed with the R2012b release from the proprietary ASCII mdl file to the (binary) zip container slx). This yielded EvoSL$^+$'s 13,919 root Simulink Git repositories with metadata. (3) Third, we use the root project metadata to similarly download all (transitive) project forks, yielding EvoSL$^+$'s 13,786 Simulink fork GitHub repositories and their metadata.

Table 6.1: Data cleaning steps: Root = project with 1+ Simulink models; License = has a license; Permissive = license allows re-distribution; MC = has 2+ model commits; ND = no duplicates; EvoSL = has model with 2+ commits.

Root	License	Permissive	MC	ND	EvoSL
13,919	2,323	2,282	1,081	1,071	924

Table 6.1 summarizes the further pre-processing, which adds to EvoSL$^+$'s metadata but for license and storage space reasons only includes the EvoSL subset (Git repositories and all metadata including issues, comments, pull requests, etc.) in the EvoSL distribution. (4) Fourth, EvoSL-Cleaner only includes an EvoSL$^+$ root project in EvoSL if the project has a license (2,323/13,919 projects) and the license allows redistribution. GitHub has a structured way for authors to set their project's license, which GitHub then converts into a corresponding license file (and subsequently exposes via an API). For the 291 project licenses GitHub did not understand (i.e., the API returns "Other"), we realized on manual review that many of them just appear to be common open-source licenses applied manually—without using GitHub's

structured license settings. We conservatively judged 250/291 projects to allow redistribution.

While for many applications (e.g., as a machine learning training set) a larger corpus is better, we also had to satisfy long term storage size limitations. We thus (5) prioritized the projects with the most changes to Simulink model files (as opposed to changes to other files). Specifically, we extract commit metadata via PyDriller [212]. While EvoSL contains the full Git repositories and all issues, pull requests, and comments, we configure PyDriller to only process the commits of each project's default-branch. We then only keep a Git repository in EvoSL if it has at least two commits across its (default-branch) Simulink model files, yielding 1,081 Git repositories with Simulink model commits.

After removing forks, the dataset may still contain other duplicates [213], which we remove heuristically. In step (6) we first mark two EvoSL Git repositories as potential duplicates if they have the same Figure 6.4 Project_Commit_Summary metric values (e.g., the same number of default-branch commits, same number of default-branch merge commits, etc.) and confirm this if they also have the same commit hashes. We keep the Git repository with the smaller GitHub project ID, yielding 1,071 Git repositories. (7) Finally, we remove Git repositories that do not change any default-branch Simulink model after that model file's initial ("check-in") commit, yielding EvoSL's 924 Git repositories.

Finally, we compare the size of EvoSL to the largest open-source Simulink corpus to date—SLNET (which does not contain project histories). To ease comparison, we focus on the 36 EvoSL projects we could open with Simulink R2019a that have the most commits of mdl/slx files ("EvoSL_{36}"). The latest version of the main branch of the projects in this EvoSL subset alone contains 714k Simulink blocks, significantly more than all of SLNET's 190k blocks in its 225 GitHub projects.

6.4.1 EvoSL Long-term Storage and Metadata

At writing we were aware of three permanent storage options that are publicly accessible, easy to cite, and offer free data deposit and download. Since Dataverse's 2.5 GB per-file limit [214] conflicts with some EvoSL projects and Figshare's free 20 GB per-user limit [215] restricts EvoSL's overall size, we upload EvoSL into Zenodo's 200 GB hard limit [199].

What EvoSL adds to the repositories (bundling, some metadata, etc.) has the permissive CC BY 4.0 license. The distribution contains EvoSL's 924 full Simulink Git root repositories (last updated in early March 2023). The size of the distribution is some 73 GB. Each of the 924 projects is in a zipped folder named after the project's GitHub project ID.

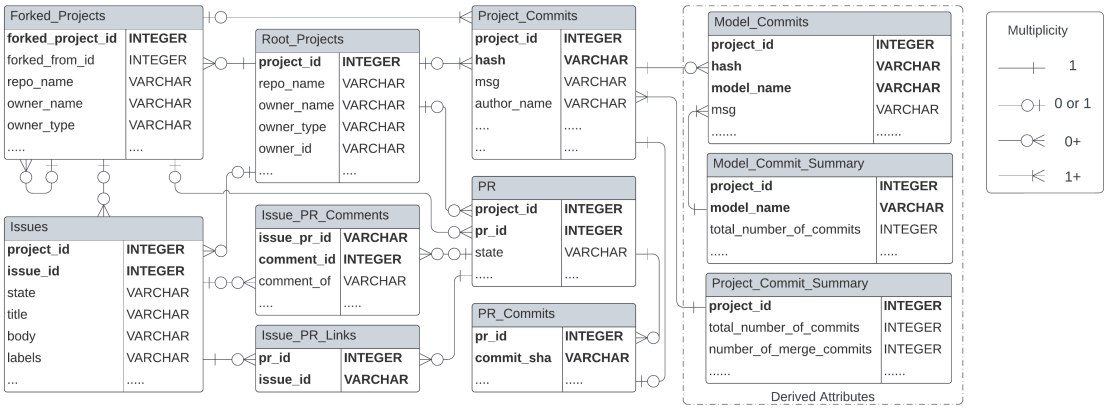


Figure 6.4: EvoSL's metadata relational database schema with multiplicity constraints, e.g.: each (default-branch) project commit is broken down into one model commit per Simulink model file change and each model commit is part of one project commit; bold = primary key; forked projects do not contain their parent project's commits (except for one initial commit).

In addition to the 924 full Simulink Git root repositories, EvoSL also contains the Figure 6.4 metadata. Specifically, the metadata is in a SQLite[1] database. The metadata mainly records the information EvoSL-Miner downloaded from the GitHub API, e.g., project popularity and engagement, issues, pull requests, and comments associated with issues and pull requests. To make it easier to select projects with certain Simulink model changes, we derive and add metadata.

First, for each project's default-branch we break down every commit that changes Simulink models into one *model commit* per model changed by that (project) commit. Specifically, we process each Git repository's default-branch commits to create one model commit per Simulink model whose slx or mdl file was touched by a given project commit (Figure 6.4 Model_Commits). A commit belonging to a merge commit is distinguished by listing more than one parent commit. We further summarize commits, e.g., the total number of default-branch commits, their number of authors, and a Simulink model's *lifetime*—i.e., the difference between a model file's first and last default-branch commit (Figure 6.4 Project_Commit_Summary and Model_Commit_Summary).

Table 6.2: Simulink root projects before (EvoSL$^+$) and after filtering (EvoSL): Issues, pull requests (PR), comments on issues and pull requests, and default-branch commits.

	Projects	Commits	Issues	PR	Comments
EvoSL$^+$ Root	13,919	419,404	5,973	7,490	14,923
EvoSL (Root)	924	143,571	3,228	1,933	10,290

[1]SQLite is widely used, free, self-contained, server-less, zero-configuration, backwards compatible, and cross-platform.

Finally, Table 6.2 compares the amount of metadata for EvoSL$^+$ and EvoSL. From a project change data perspective, EvoSL is clearly an interesting (but non-representative) sample of the full EvoSL$^+$ root projects. For example, while EvoSL$^+$ contains over 15 times of the root projects of EvoSL, EvoSL contains over one third of all EvoSL$^+$ default-branch project commits and over two thirds of all EvoSL$^+$ issue and pull request comments.

6.4.2 Overview of EvoSL's Simulink Model and Project Changes

It is well-known that for the entirety of GitHub the distribution of commits over projects is heavily skewed toward a few very active projects, with a long tail of projects having under 50 commits [9]. It is thus no surprise that GitHub's Simulink projects default branches follow a similar long-tail distribution (Figure 6.5). For example, in EvoSL$^+$ the median number of default-branch project commits is six and the median number of model commits per project default-branch is one. Further, 91% of projects have under 50 total default-branch commits and 56% of projects only have one model default-branch commit.

To better understand model change timing and the share of Simulink models that is changed during the project, Figure 6.6 breaks each project default-branch's duration into 10 buckets of equal length (normalized to each project default-branch's duration). Here project duration is the duration from a project default-branch's first to last commit as recorded by the timestamps assigned by the committers' machines. While this approach has its pitfalls, the more-active projects are usually less affected and we performed the basic recommended sanity checks to ensure there are no impossible outliers (e.g., commits with Unix time zero) [182].

Specifically, if a project only has one default-branch commit then Figure 6.6 assigns this commit (only) to the last bucket (90–100%). This explains Figure 6.6a's

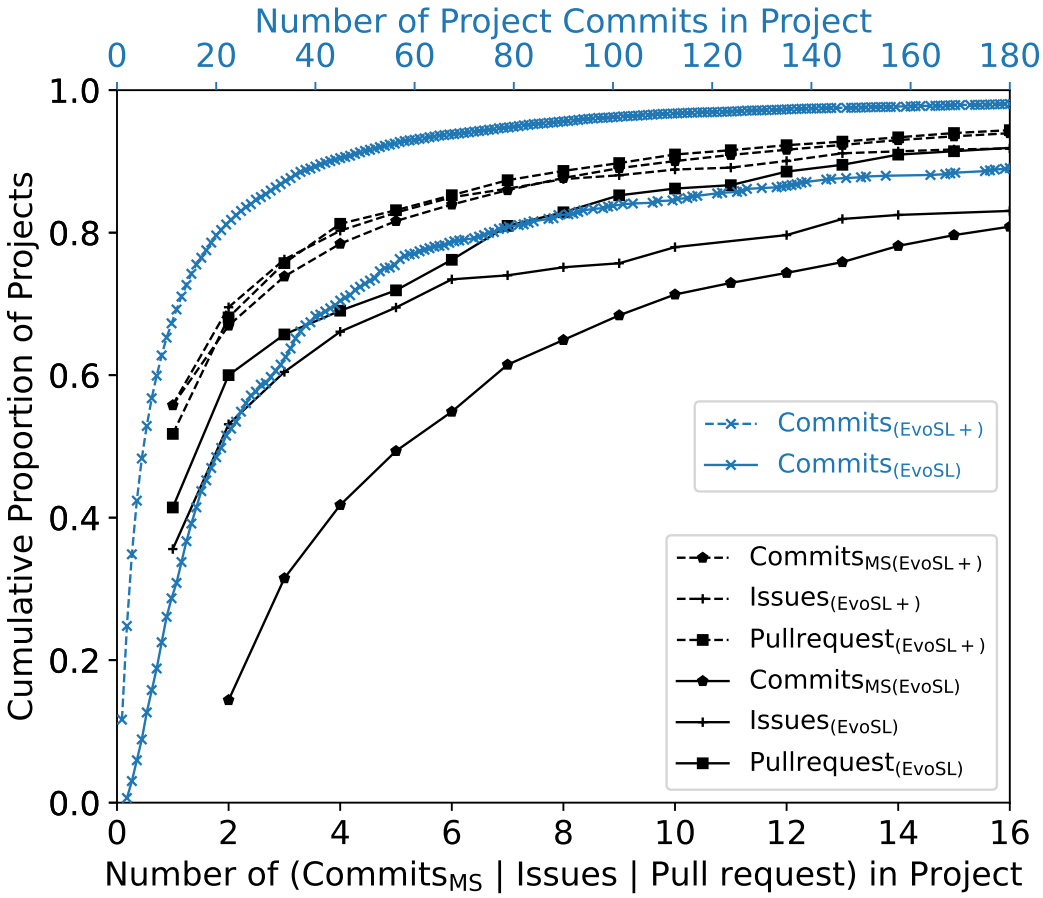


Figure 6.5: Root project percentage (y-axis) with up to the given number of default-branch commits, default-branch commits of 1+ mdl/slx files, issues, and pull requests (x-axes).

spike in the 90–100% bucket, as EvoSL$^+$ is skewed towards projects with few default-branch commits. Similarly, the spike in its 0–10% bucket says that many EvoSL$^+$ projects commit (or "dump") much of their changes (and especially changes to Simulink model files) together around the time of the initial default-branch commit.

The last-bucket concentration of commits, commits of 1+ mdl/slx files ($\text{commits}_{\text{MS}}$), and share of mdl/slx files in a commit ("models under development") all decrease in EvoSL, again when looking at the subset of EvoSL projects with 10+ default-branch $\text{commits}_{\text{MS}}$ (Figure 6.6c), and yet again when looking at the 36 top-$\text{commits}_{\text{MS}}$ EvoSL

projects we could open with Simulink R2019a (Figure 6.6d). At the same time, these measures trend upward for the non-start/finish buckets. For example, for EvoSL projects with 10+ default-branch $\text{commits}_{\text{MS}}$, each bucket contains commits that overall include at least 10% of the default-branch mdl/slx files.

Table 6.3: Default-branch metrics: Commits, commits per day during project duration, merge commits ($\succ$), and commits of 1+ mdl/slx files (MS); commit authors and $\text{commit}_{\text{MS}}$ authors; l = low; h = high; med = median; std = standard deviation.

	EvoSL^{+} (root)					EvoSL (root)					EvoSL_{36} (root)				
Default-branch:	l	h	avg	med	std	l	h	avg	med	std	l	h	avg	med	std
Commits	1	15,060	30	6	305	2	15,060	155	22	992	102	1,452	499	381	349
Commits / day	0	91	2	1	4	0	50	1	0	3	0	1	0	0	0
$\text{Commits}_{\succ}$ [%]	0	55	2	0	6	0	48	5	0	7	0	17	9	9	5
$\text{Commits}_{\text{MS}}$ [%]	0	100	43	33	31	0	100	36	31	25	6	88	35	27	19
Authors	1	103	2	1	3	1	103	4	2	8	1	45	14	11	11
$\text{Authors}_{\text{MS}}$ [%]	1	100	84	100	25	2	100	73	67	28	25	100	62	56	21
Durations [days]	0	5,909	116	4	326	0	5,909	443	135	703	264	5,909	1,553	1,207	1,076

The Figure 6.6 distributions over normalized project lengths further resemble traditional software development projects (rather than file dumps) when putting them into context of the Table 6.3 project metrics. First, EvoSL's, EvoSL^{+}'s, and EvoSL_{36}'s absolute project lengths range up to 16 years, with average project lengths of 116, 443, and 1,553 days and median project lengths of 4, 135, and 1,207 days. Finally, EvoSL and EvoSL_{36} projects have a median of two and 11 authors, of which more than half also commit slx/mdl files.

6.5 Replicating an Industrial Study With EvoSL

Our main contribution is the EvoSL corpus itself, as it allows exploring various research questions on Simulink model and project changes, including commits, GitHub issues, pull requests, other project metadata, and their correlations. While open-source projects will never be exactly like industrial closed-source development

in all aspects, **here we are asking if studying open-source projects can yield results that are comparable to studying closed-source Simulink projects**.

At a minimum, this would allow the research community to develop hypotheses and tools that could then be tested and validated more easily in an academic-industrial collaboration. This may lead to faster progress than relying on all the heavy lifting of developing hypotheses and tooling from scratch being done in closed-source academic-industrial collaborations.

To explore this question, we pick one recent representative empirical study of the changes in a closed-source Simulink project and replicate the study using EvoSL projects. Jaskolka et al. [4] examined changes across different model versions of a proprietary industrial software repository of an automotive control system to understand how Simulink models evolve over time. Their analysis shows that well-accepted software engineering principles (such as low degree of change to interfaces) are not practiced and engineers spend significant amounts of time in non-value added work such as migrating the project to new Simulink versions. Specifically, we are investigating the following three research questions, all copied verbatim from Jaskolka et al. [4].

RQ1 What basic elements change the most?

RQ2 Which blocks are involved in changes most frequently?

RQ3 Which are identified categories of change?

6.5.1 Experimental Setup Following C-study

To replicate the earlier study [4] (which we call C-study, where C may stand for closed-source or car industry), we try to follow C-study's setup and procedures as closely as possible, using both the same conceptual framework of Simulink model changes and tooling for collecting change data.

For the conceptual framework, Figure 6.7 summarizes the relevant parts of C-study's meta-model of Simulink models. Here a block diagram is composed of six *element* types—block, line, port, mask, annotation, and configuration. For each of these six element types, C-study collects four change types—add, delete, rename, and (otherwise) modify.

C-study detects a change by comparing two snapshots of a model. An *added* element does not exist in the before- but in the after-model. A *deleted* element exists in the before- but not in the after-model. A *renamed* and *modified* element each exist in both the before- and the after-model but have their name or another parameter changed.

On the tooling side, Figure 6.8 gives an overview of how we adapted C-study's model change computation. C-study queried a Rational Synergy commercial issue tracking system to extract before- and after- model file versions from a Rational Change commercial change management system [216, 217]. Given EvoSL's use of standard Git repositories and its inferred model commit metadata, it is straightforward to similarly extract such before- and after- model file versions from EvoSL.

Since we do not know how C-study treated merge commits (or commits from non-default branches) we focus on non-merge default-branch commits. C-study's open-source Model Comparison Utility [4, 5] passes each pair of before- and after- file versions to Simulink's built-in model comparison tool and breaks its output down into individual model element changes. Due to Simulink API limitations, C-study's Model Comparison Utility discards non-functional changes (such as layout) and block defaults. We store and distribute the remaining derived element changes in a SQLite database.

For this replication study we had to make a trade-off between model selection and replication accuracy. The key reason is that using two different Simulink ver-

sions and their built-in model comparison tools on the same before- and after- model pair can produce vastly different results, even when both Simulink versions directly support the model file versions.

Concretely, as C-study used Simulink R2019a we passed random Simulink models developed with R2019a to both Simulink R2019a and the (ostensibly backward-compatible) R2022b. Despite the tool documentation being silent on this, about a quarter of changes were only reported by one of the two Simulink versions, with no report being a superset of the other. As we cannot run R2022b on the closed-source C-study repository we are stuck with Simulink R2019a and (as Simulink is not directly forward-compatible) we exclude from this study model files developed with R2019b or later that Simulink R2019a refuses to open.

6.5.2 Simulink Model Changes From EvoSL Sample: EvoSL_{36}

As C-study focuses on Simulink model changes we pick the 50 EvoSL projects that have the most default-branch changes to mdl/slx files ("$\text{commits}_{\text{MS}}$"). Due to our experimental setup constraints, we remove 9 (newer) projects Simulink R2019a cannot open. We remove five additional projects who have a subset of the default-branch $\text{commits}_{\text{MS}}$ of another project. (EvoSL removed all explicit fork projects and exact non-fork duplicate project histories but not such non-forked almost-duplicates.) We are thus left with 36 EvoSL projects ("EvoSL_{36}"). Following C-study, we ran its Model Comparison Utility [4, 5] on each default-branch before- and after-commit model pair of EvoSL_{36}, yielding 590,300 Simulink model changes (C-study analyzed 2.8M such changes).

Table 6.4 puts EvoSL_{36} in context of C-study's published basic project characteristics [4]. While EvoSL_{36} has default-branch changes in fewer mdl/slx files (some 900 in total vs 3,945), has fewer total default-branch $\text{commits}_{\text{MS}}$ in a sin-

Table 6.4: Basic project metrics copied from C-study [4] and EvoSL_{36}'s default-branch distributions; l = low; h = high; med = median; std = standard deviation; m = model size.

	[4]	EvoSL_{36} (default branches)				
		l	h	avg	med	std
Largest m. [blocks]	37,814	6	51,655	2,368	357	8,642
Avg m. [blocks]	1,200	5	8,366	424	113	1,438
$\text{Commits}_{\text{MS}}$	1,354	60	598	141	111	105
Duration [months]	75	9	197	52	40	36
Changed model files	3,945	1	176	25	13	37

gle project (598 vs 1,354), and has a lower average default-branch model size (424 vs 1,200 blocks), on all the measures, except changed model files, EvoSL_{36} is within the same order of magnitude as C-study. On the flip side, EvoSL_{36} has a larger maximum default-branch model size (52k vs. 38k blocks), longer maximum default-branch project duration (16 vs. 6 years), and more total default-branch commits with mdl/slx file changes in total (5k vs 1,354 $\text{commits}_{\text{MS}}$)—again all within the same order of magnitude.

To further gauge EvoSL_{36}'s suitability, we checked the criteria recently laid out with the help of a Simulink expert [89] when analyzing the suitability of an earlier corpus [31]. Of these criteria, we did not reach a conclusion on a steady increase of commits towards project end (as we do not know EvoSL_{36} projects' future timelines) and a non-low percentage of commits being merge commits (it is unclear if EvoSL_{36}'s median 9% of commits meets this bar).

That paper's remaining provided metrics and example values [89] all largely align with EvoSL_{36} (Table 6.3), i.e., a high project duration (2.3k vs. EvoSL_{36}'s median 1.2k days in the default-branch), many authors (16 vs. median 11), many project commits (589 vs. median 381), and many default-branch commits affecting mds/slx files (44% vs. median 27%). Together with our manual sampling of commit

messages, we conclude that EvoSL$_{36}$ projects are not synthetic outliers generated by a random generator, but represent suitable human development activity.

6.5.3 RQ1: What Basic Elements Change the Most?

Breaking EvoSL$_{36}$'s 590,300 default-branch element changes down by element and change type as C-study did yields Table 6.5's right columns. To ease comparison, we normalize each element type's change count (e.g., EvoSL$_{36}$'s 30k block renames) by dividing that element type's total changes (e.g., EvoSL$_{36}$'s 300k total block changes—yielding EvoSL$_{36}$'s 10.1% block rename rate).

Table 6.5: Types of Simulink model element changes in C-study and EvoSL$_{36}$ (default branches); normalized = element type's specific changes divided by that element type's total changes; Re = rename; Mod = modify; Del = delete; EC/TC = element type's total changes divided by total changes; Anno = annotation; Conf = configuration.

	C-study (normalized)					EvoSL$_{36}$ (normalized)					EvoSL$_{36}$ (absolute)				
	Re	Mod	Del	Add	EC/TC	Re	Mod	Del	Add	EC/TC	Re	Mod	Del	Add	Total
Block	12.0	24.9	22.9	40.2	55.3	10.1	35.4	23.4	31.1	50.7	30,183	106,093	69,985	93,295	299,556
Line	0.2	2.0	39.2	58.6	38.9	0.2	1.8	43.3	54.6	41.6	558	4,385	106,575	134,340	245,858
Port	0.3	27.6	27.6	44.5	3.2	0.0	0.5	43.4	56.1	4.2	0	124	10,774	13,951	24,849
Mask	0.0	19.8	16.9	63.2	1.8	0.0	43.7	23.7	32.5	1.2	0	3,148	1,709	2,343	7,200
Anno	4.0	10.4	34.2	51.4	0.8	6.3	7.4	44.3	42.0	1.1	407	482	2,871	2,723	6,483
Conf	0.0	98.5	0.0	1.5	0.0	2.2	65.0	3.5	29.3	1.1	142	4,130	220	1,862	6,354
All	6.7	15.9	29.4	48.0	100.0	5.3	20.1	32.5	42.1	100.0	31,290	118,362	192,134	248,514	590,300

C-study makes several observations about this element change breakdown and draws two main conclusions, all of which could equally be done with EvoSL$_{36}$'s corresponding (default-branch) data, as follows. (1) First, C-study observes that the most frequently changed element type is blocks, which aligns with 300k of 590k total EvoSL$_{36}$ changes being blocks. (2) Second, C-study finds that line changes follow closely behind block changes, which again aligns closely with EvoSL$_{36}$'s 246k line vs. 300k block changes.

(3) Third, C-study notices that line changes are dominated by first add (59%) and then delete (39%), which similarly occurs in EvoSL_{36} (55% add and 43% delete). Combined with add- and delete-line having higher absolute numbers than add- and delete-block, C-study explains how replacing a block triggers a deleting and adding a line.

(4) Fourth, not directly referencing any additional data, C-study determines that a similar dynamic is at play with ports. As all of the previous data observations are equally true in EvoSL_{36}, we could have equally used EvoSL_{36} to conclude to omit both line and port changes from further analysis. (5) Finally, C-study observes that masks, annotations, and configurations have the least changes (some 3% overall), which again aligns with some 3% in EvoSL_{36}.

Beyond C-study's observations, there are several other similarities. For example, in both data sets the most common change type is add, followed by (in order) delete, modify, and rename. The one big outlier in both datasets is configuration, which is dominated by modify and followed by add. In both datasets blocks have the highest rename rate, followed by annotations, and lower rates for the remaining element types.

Finding 8: *Besides additional similarities, all observations C-study makes about its change data are equally true for* $\textit{EvoSL}_{36}$*. We thus assume researchers could draw the same conclusions C-study drew.*

6.5.4 RQ2: Most Frequently-changed Block Types

To analyze which blocks change most frequently, C-study aggregates blocks by their block type (as given by the block's BlockType parameter). C-study's five block types with the most changed block instances are Inport (11.8% of all C-study

changes), Outport (9.2%), From (5.4%), Constant (4.4%), and SubSystem (4.2%). These five block types are also in EvoSL$_{36}$'s top-six block types by most block instance changes. The one exception is the Reference block type, which dominates EvoSL$_{36}$ likely because it represents custom blocks we did not load from one of EvoSL$_{36}$'s custom libraries.

Figure 6.9 shows all block types whose block instances have over 50 changes across the EvoSL$_{36}$ default branches. The frequency of the six block types with the most block instance changes are Reference (16% of all EvoSL$_{36}$ changes), SubSystem (7.3%), Inport (4.9%), Outport (3.5%), Constant (1.8%), and From (1.5%).

Besides Reference, EvoSL$_{36}$ also has a higher rate of Subsystem changes. Subsystem blocks are typically used to modularize and organize a large model into smaller and more manageable components. We assume C-study's slightly lower rate of Subsystem block changes stems from each project tending to become relatively more stable over time. This progression playing out for each EvoSL$_{36}$ project would explain EvoSL$_{36}$'s overall higher rate of such structural changes. EvoSL$_{36}$'s next two most frequently changed block types, Inport and Outport, appear in this order also in C-study.

Finally, we examine the ordering of block types by most-changed total block instances with Kendall's rank and p-value less than 0.05. Based on this test, the trend of most-to-least frequently changed block types in C-study and EvoSL$_{36}$ are strongly positively correlated ($\tau = 0.99$).

Finding 9: *EvoSL$_{36}$ mimics several characteristics of C-study's block type change distribution, including the most-changed block types and a strong correlation between the order of block types by block changes.*

6.5.5 RQ3. Which Are Identified Categories of Change?

Table 6.6: C-study's 13 Simulink block categories [4].

Category	Example Blocks
(Model) Interface	root-level Inport/Outport, global DataStoreRead/DataStoreWrite, FromFile/ToFile, FromSpreadsheet, ToWorkspace/FromWorkspace
Signal Routing	non-root Inport/Outport, Goto/From, local DataStoreRead/Write, BusCreator, Merge, Assignment
Signal Attributes	RateTransition, DataTypeDuplicate, SignalConversion, DataTypeConversion
Structural	SubSystem, Reference
Conditional	If, Switch, SwitchCase, ManualSwitch
Discrete	Delay, UnitDelay, Filter, Integrator
Math	Sum, MinMax, Rounding, Abs, Gain
Logic	RelationalOperator, LogicalOperator
Trigger	TriggerPort, EnablePort, ActionPort
Sources	Ground, Step, Clock, Constant
Sinks	Terminator, Scope, Display
Documentation	ModelInfo, DocBlock
Custom	S-Function

The standard Simulink language block libraries categorize blocks pertaining to their purpose or a common quality. However, block types in these groups overlap with the other groups. To categorize each block type to a non-overlapping category, Jaskolka et al. created a new category scheme in which each block type falls under a single category according to their purpose. They also introduced new categories such as Documentation and Interface. Table 6.6 shows the list of categories with some example blocks, and full details can be found in their work [218]. We adopted the new category scheme and analyzed the changes according to it. Block types not listed in Jaskolka's category scheme we marked as "Others".

Table 6.7: $EvoSL_{36}$'s block changes by C-study's Table 6.6 categories; Others = all newer and uncategorised blocks types.

Block Category	%	Block Category	%
Structural	49.55	Signal Routing	28.90
Math	6.18	Sources	3.96
Others	3.10	Discrete	2.80
Sinks	2.08	Signal Attributes	1.03
Conditional	0.97	Logic	0.85
Custom	0.75	Interface	0.62
Trigger	0.19	Documentation	0.01

Table 6.7 shows the ratio of (default-branch) block changes by block category. Most $EvoSL_{36}$ changes are on structural blocks, followed by signal routing, math, and source blocks. Unlike C-study where interface changes contributed to over one-third of block changes, in $EvoSL_{36}$ interfaces are stable with under 1% of block changes, indicating good modeling practices. We delve into a few categories below.

6.5.5.1 Changes to Signal Routing and Structural Blocks

In Simulink, data produced and processed by blocks is routed via signal lines. Rather than analyzing the signal lines (as discussed in Section 6.5.3), analyzing changes to the blocks that are responsible for routing, combining, creating, and selecting data is more revealing. Table 6.7 shows that engineers spend a substantial amount of time managing signal data, as 29% of block changes are signal routing changes.

The many changes to signal routing blocks go hand in hand with the most frequently changed block category, i.e., Structural. In model based development, complexity is abstracted through creating hierarchical models. The many changes to SubSystem and Reference blocks (Figure 6.9), contributed to significant changes to signal routing between the models' hierarchical layers.

6.5.5.2 Changes to Documentation Blocks

Simulink provides various options for documenting models. Similar to code comments for understanding textual programs, in Simulink one can use annotations including text and images embedded in the model. Users can also embed plain text or a document via a *DocBlock* [219], which may contain a more thorough description of the design. Finally, Simulink's *Model Info* [220] block shows (automatically updating) revision control information such as creator and the last modified date.

Compared to block changes overall, EvoSL_{36} has significantly fewer changes to documentation-related blocks. The low documentation change frequency is inline with existing software documentation for Simulink models [221] and most other software systems [222,223]. 99% of the documentation-related changes are made through annotations. Unlike in industrial development where Model Info is encouraged to keep track of the original model's creator and other metadata, the EvoSL_{36} open-source projects do not follow the practice.

> ***Finding 10:*** *As in C-study, EvoSL_{36} engineers made much more changes to signal routing and structural blocks than to implementing algorithms, leading to the same key conclusion as C-study.*

6.6 Threats to Validity

One threat to validity is that we do not know if C-study analyzed model changes from production and all development branches. As C-study does not mention branches at all [4,218], our replication focuses on model changes that are either direct commits to the default branch or added to the default branch via merging. By skipping analysis of commits from branches not ultimately merged back into the default branch (including those that got removed via squashing before merging to the default

branch), we missed 1,084 $\text{commits}_{\text{MS}}$ (and their 2,540 model commits), beyond the 5,070 $\text{commits}_{\text{MS}}$ (and their 9,845 model commits) we analyzed for EvoSL_{36}.

EvoSL is curated from GitHub and may not be representative of all open-source Simulink repositories. GitHub does not recognize Simulink as a programming language. We did a thorough search via GitHub's API, filtering GitHub projects written in MATLAB on top of a "Simulink" keyword search. Note that projects that only contain Simulink models are not tagged with a programming language, so our search may have missed them. It is impractical to search 330 million GitHub repositories to filter Simulink projects.

C-study's Model Comparison Utility captures Stateflow block changes in the model snapshots but did not label any of the EvoSL_{36}'s change with Stateflow. To assess EvoSL's potential to facilitate research on Stateflow changes, we ran the utility using MATLAB/Simulink 2022b on EvoSL, which yielded no Stateflow-related changes. Our attempts to identify relevant projects were also unsuccessful, despite finding at least 15 projects that mentioned Stateflow in their descriptions in our metadata. While EvoSL may not be suitable for Stateflow change studies, a few EvoSL projects could still contain Stateflow-related blocks. Also, we provide the EvoSL element change data with its metadata for further analysis.

6.7 Related Work

Simulink models have largely been curated with manual or semi-automated approaches [22, 31, 89, 169]. Sánchez et al. used Google's BigQuery to filter and extract the largest open-source Simulink models to test their tool [169]. Chowdhury et al.'s SLforge project performed the first large-scale study of 391 Simulink models, whose primary focus was to test the Simulink tool chain [22]. The authors later extended their corpus to 1071 publicly available models [31]. Boll et al. reproduced

the corpus developed by Chowdhury et al., providing deeper insights on the models and modeling characteristics [89]. Shrestha et al. pioneered fully automated mining of some 400 non-hierarchical models to train a deep learning model for Simulink tool chain testing and later curated the first self-contained corpus of Simulink models, addressing limitation of earlier corpora [85,198]. Unlike EvoSL, none of these corpora offer projects with full revision history.

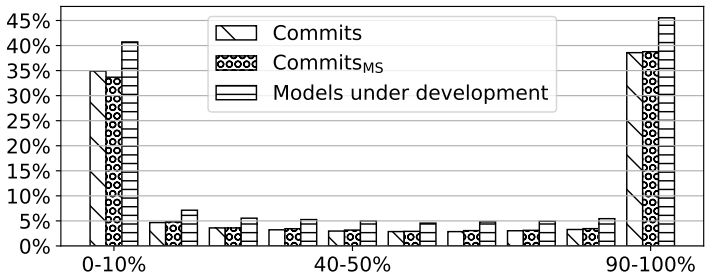
Existing work in the field of model evolution is focused on clone detection, variant management, and studying the synchronous co-evolution of models and tests. To detect clones, Stephen et al. developed the SIMONE algorithm, which has been used to track the evolution of model clones and refactor cloned fragments into a library [87, 195]. Haber et al. proposed delta operations, including add, remove, modify and replace elements, to obtain desired variant models [224]. Schlie et al. focused on improving variability mining approaches by adding blocks and hierarchical levels [165]. Rapos et al. employed Simulink's built-in comparison tool to extract change information and investigate the synchronous co-evolution of models and tests in closed-source industrial models [196]. Jaskolks et al.'s case study stands out for its comprehensive classification of changes to model elements, which we replicated in this study using EvoSL.

Mining source code from software repositories has yielded rich information researchers have leveraged for code-based research [225–227]. GitHub especially has emerged as a primary source of open-source repositories for empirical research. Consequently, researchers have developed several tools to facilitate mining from GitHub [97, 212, 228, 229]. In this study, we used PyDriller and PyGitHub to curate EvoSL [97, 212]. In recent years, there has been increasing interest in mining model-based artifacts. The MAR search engine [230,231] has been developed to facilitate model-driven engineering efforts, i.e., the tool searches existing corpora for Simulink models. On

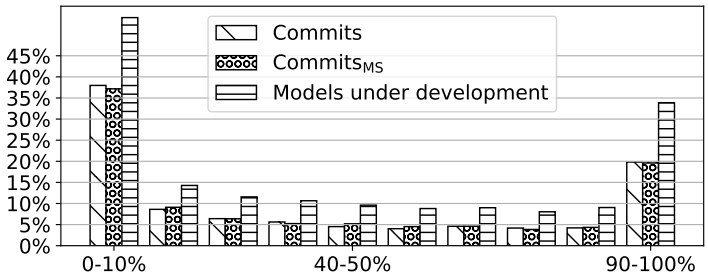
the other hand, tools such as ModelMine allow for artifact searches directly from GitHub by searching based on file extensions. However, the tool incorrectly labels the Simulink model file extension as ".simulink".

6.8 Conclusions

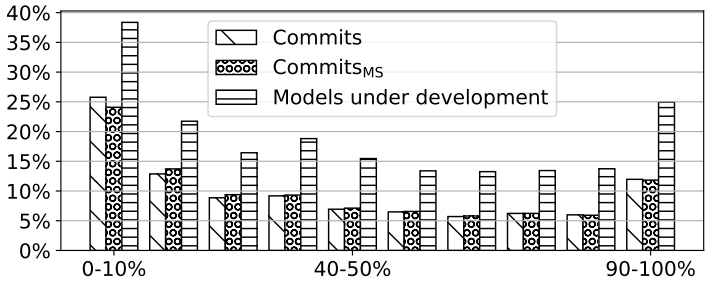
In this study, we emphasize the importance of readily accessible corpora for performing replication, reproduction, extension, and verification studies. Several safety-critical industries use MATLAB/Simulink as a standard tool for system modeling and analysis, necessitating large-scale model evolution studies. However, there has been no readily accessible corpus for such studies. To address this gap, we introduced EvoSL as the first large corpus of Simulink projects, including model and project changes, which is available under a permissive open-source license and included its collection and analysis tools. On a EvoSL subset we successfully replicated a case study of model changes in a closed-source industrial project.



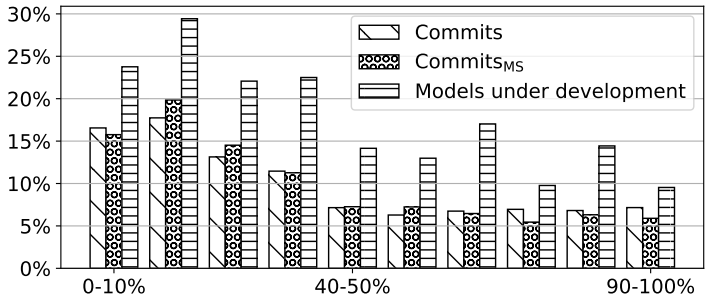
(a) All 13,919 EvoSL^{+} Simulink root projects.



(b) All 924 EvoSL Simulink (root) projects.



(c) The 292 EvoSL Simulink (root) projects with 10+ commits$_{MS}$.



(d) The 36 EvoSL$_{36}$ (root) projects.

Figure 6.6: Across projects' normalized duration on x-axis: Total default-branch commits, default-branch commits$_{MS}$, and percentage of mdl/slx files included in bucket's commits.

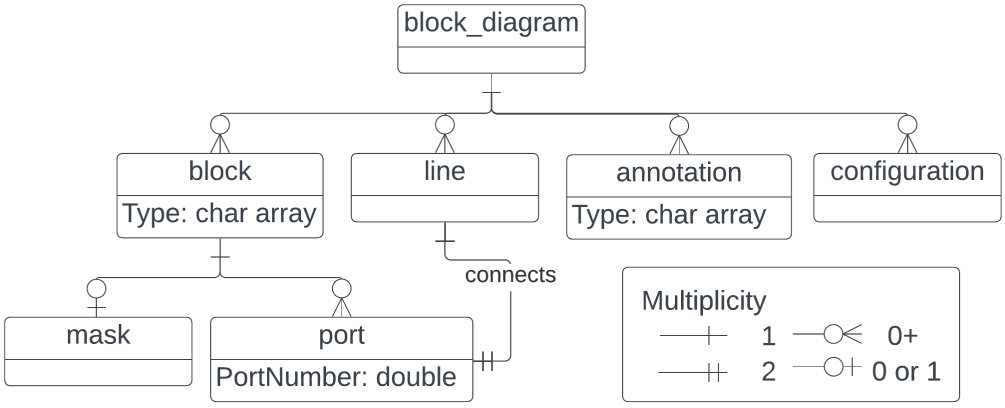


Figure 6.7: Simplified Simulink meta-model: 6 element types [4].

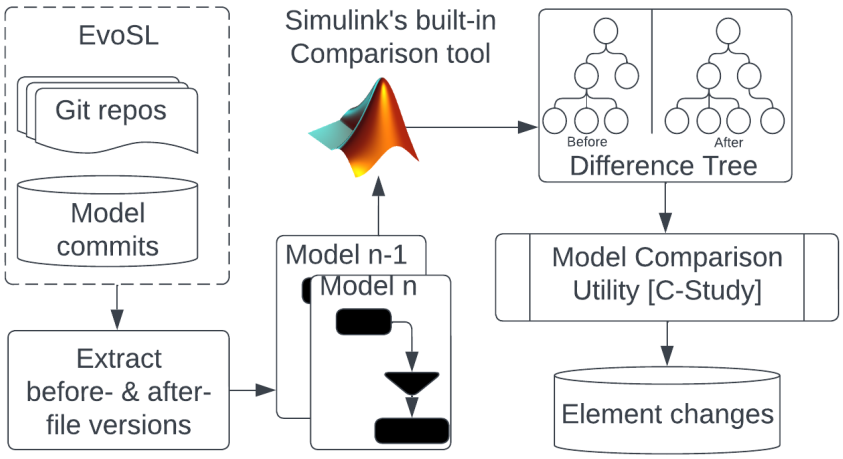


Figure 6.8: Using C-study's Model Comparison Utility [4, 5] to mine Simulink model changes in EvoSL.

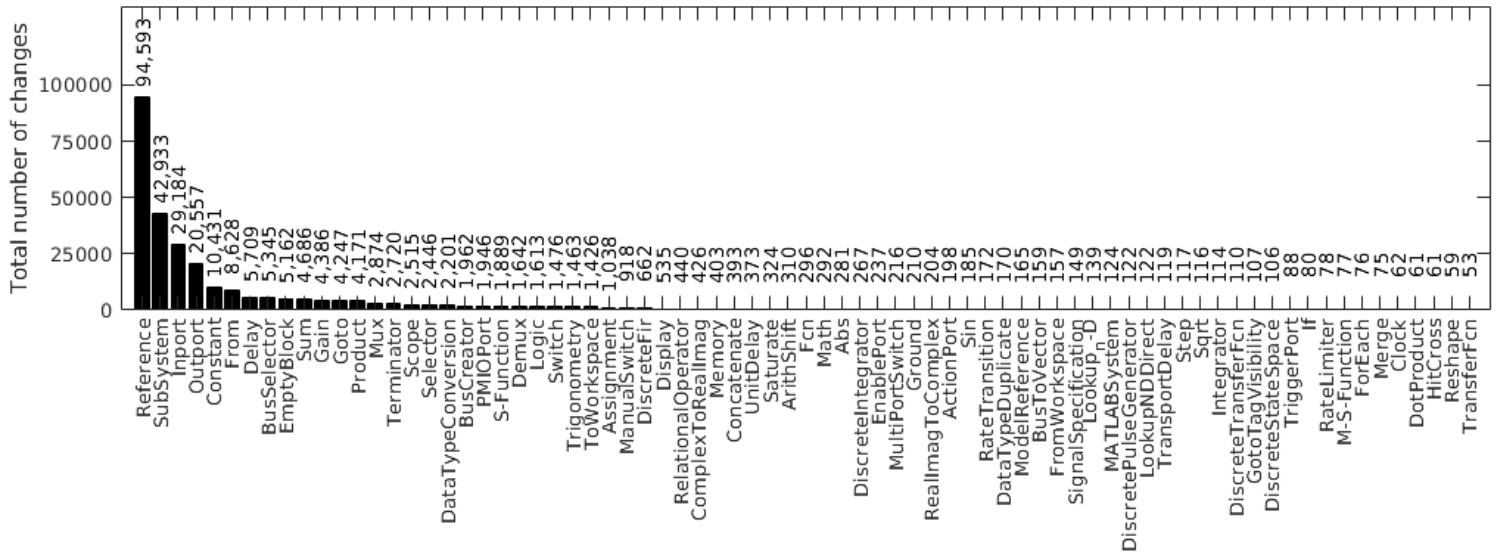


Figure 6.9: EvoSL$_{36}$'s 299,556 (default-branch) Simulink block changes by block type (showing block types with 50+ changes).

CHAPTER 7

ScoutSL: An Open-source Simulink Search Engine

This chapter was originally published in 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), Västerås, Sweden, 2023, pp. 70-74, doi: 10.1109/MODELS-C59198.2023.00022. It is reproduced here with permission from IEEE without revision [34].

7.1 Abstract

Simulink is one of the most widely used modelling languages in safety-critical industries. Most models created in industrial settings are valuable intellectual property to their companies and are thus often not publicly available. But to study Simulink software engineering processes or to develop new Simulink tools, access to models with relevant properties is vital for researchers. We conducted a community survey to find out what kind of models and model metrics are of interest to researchers. With these results, we implemented ScoutSL (`http://scoutsl.net`), a tool that gives researchers easy online access to over 100k open-source Simulink models from which they can select a subset according to their needs. A short video demonstration is available online at `https://youtu.be/HwsHL8LrVCM`

7.2 Introduction

Searching for Simulink models presents challenges due to the absence of a convenient method for finding such models beyond text-based searches. Traditional textual programming language search attributes such as lines of code do not apply to Simulink

models, which are developed via graphical block diagrams. So the requisite search attributes for Simulink models are not adequately addressed.

Despite the proliferation of open-source repositories that have accelerated empirical study of code [232,233], there is a dearth of such studies on MATLAB/Simulink. This can be attributed to the lack of easily accessible model corpora and user-friendly tools that cater to novice users. Researchers have encountered difficulties in discovering third-party Simulink models suitable for utilization in their studies, particularly for stress testing their developed tools or validating the generalizability of novel techniques they are attempting to address. The absence of an easily accessible and user-friendly tool remains the primary hindrance [28, 112, 119, 176, 198, 234, 235].

Popular code hosting platforms such as GitHub and GitLab lack the capability to filter attributes specific to Simulink models, and the identification of Simulink projects is challenging as these platforms do not label projects with Simulink as a programming language. Moreover, utilizing the APIs of these platforms for research purposes is time-consuming due to API rate limits. For instance, GitHub's API restricts authenticated users to 30 search requests per minute, yielding 1k results per request and 5k other requests per hour. Considering that GitHub currently hosts over 330 million repositories, obtaining results, excluding downloading and further analysis, would require at least 330k requests (around 180 hours) [236, 237].

Few existing model-based search tools, like MAR [238], require a deep understanding of the metamodel, while others, such as ModelMine [239], offer a user-friendly search engine but rely on the GitHub API, which inherently imposes limitations on the number of search results it can retrieve. Furthermore, GitHub is not the sole source of Simulink projects. Simulink vendor MathWorks also provides a platform, which serves as a repository for community-developed projects.

Recent efforts on developing large collections of Simulink models have focused on carefully curating corpora of Simulink models manually [31] and later automatically [240] and maintaining metadata of commonly used attributes. Such corpora are either maintained in non-permanent locations though or packaged as a single non-divisible set, making them difficult to sample [240, 241]. Downloading a large corpus to sample a small subset of models is also often inconvenient.

To gain insight into attributes of Simulink models that are of interest to the research community, we conducted a survey involving researchers. The survey confirmed their struggles in getting suitable (e.g., size, publishable) models for their research as seen in Figure 7.2. Consequently, we developed a web-based search tool that allows users to easily sample models. Our tool, ScoutSL, expands upon the existing SLNET [240] and EvoSL [194] infrastructure to extract Simulink project attributes, collect model metrics and compute derived metrics. The tool offers advanced fine-grained filtering attributes, enabling users to efficiently sample desired models. We have indexed over 18k projects containing more than 100k Simulink models. To the best of our knowledge, ScoutSL is the first tool specifically designed for searching Simulink projects and models, offering filtering attributes not available through other search engines. To summarize, the paper makes the following major contributions.

- Our survey results show that researchers often struggle to get relevant models for their research and would likely benefit from a Simulink search engine.
- We developed a Simulink search engine deployed in a tool called ScoutSL whose search interface and ranking scheme are based on survey responses.
- The tool and all artifacts are open-source [242, 243].
- The search engine is accessible through its web component available online at `http://scoutsl.net`.

7.3 Background: Simulink, SLNET, and EvoSL

Simulink is a cyber-physical system (CPS) design and simulation tool that is a de-facto standard in many safety-critical industries. Engineers design a CPS as a model that contains interconnected blocks, where each block may accept data, perform some operation on the data, and transmit its output to other blocks, as depicted in Figure 7.1. Simulink provides an extensive library of blocks and toolboxes to design and simulate complex multi-domain systems.

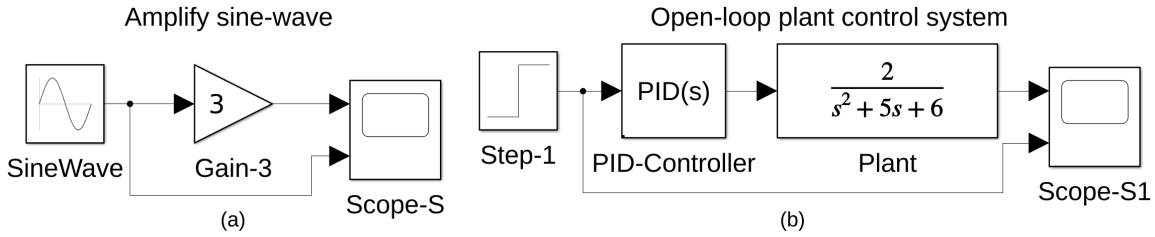


Figure 7.1: Two tiny example Simulink models.

To enable empirical studies on Simulink models, researchers have curated large corpora of open-source Simulink models. The most extensive corpus available to date is SLNET [240], which contains Simulink models from two popular hosting sites. SLNET has 3k Simulink projects with their 8k Simulink models (excluding library and test harnesses), collectively featuring over 1M blocks. Boll et al. [89] confirmed large open source corpora to be suitable for empirical research. SLNET is complemented by mining and metric tools. However, as SLNET primarily consists of Simulink model snapshots, it does not support evolution studies.

To address this issue, EvoSL [194] extended SLNET-Miner and curated Simulink repositories from GitHub. EvoSL consists of 924 projects with over 140k default-branch commits. SLNET and EvoSL are self-contained and redistributable.

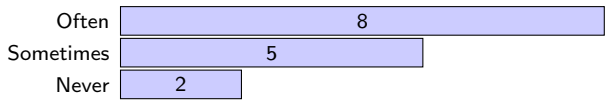


Figure 7.2: Responses to "Do you have difficulties finding adequate Simulink models or projects for your research?"

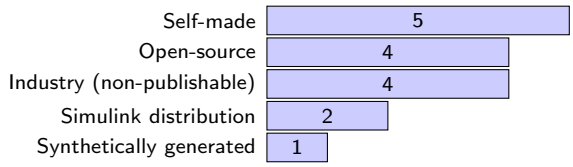


Figure 7.3: Responses to "Where do you usually obtain your Simulink artifacts from?"

7.4 Survey of Simulink Users

We aimed to assess the potential need for a Simulink model search engine by asking Simulink researchers. We then used the survey results to develop ScoutSL.

From a literature review of Boll et al. [244] we extracted 215 academic papers' co-authors that report on Simulink tools and their empirical evaluation. In July and August 2022 we invited them to our Google Forms based anonymized online survey. 16 researchers participated in our survey, from which we discarded one participant (who responded "not applicable" to every question), leaving 15 participants. While all questions and responses are available online [243], we provide a brief summary of the questions and responses in the sequel.

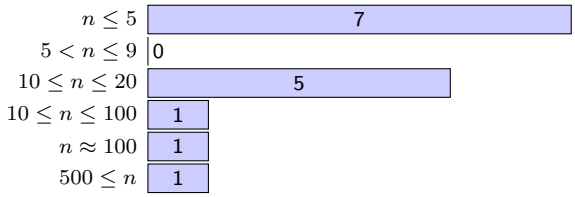


Figure 7.4: Responses to "How many models would you need for your typical research project?"

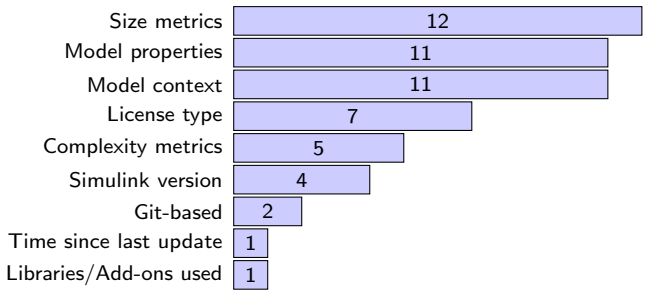


Figure 7.5: Responses to "What are Simulink model metrics that are relevant for your research?"

In our first question of the survey, we asked the researchers, "what is the purpose of Simulink models in your research?" Of 15 participants, six reported that their main use-case for Simulink models was tool evaluation. Other use-cases like scalability evaluation, performance evaluation, model co-simulation, industrial process modelling, prototyping, test automation, testing, verification, code generation optimization, compilation, model deployment, and replication were mentioned by one participant each. All the following questions and their detailed results are shown in Figures 7.2 to 7.7.

Over 85% of participants (13/15) faced difficulties finding appropriate models for their research projects (cf. Figure 7.2). When it came to acquiring models, one third of the participants created their own Simulink artifacts, followed by using closed-source and open-source models, see Figure 7.3. Figure 7.4 illustrates that the majority of participants said they require 20 or fewer models for their research—the unconventional ranges follow the responses. Figure 7.5 breaks down the model metrics of interest reported by the participants.

In response to our questions regarding the adoption of open-source Simulink models, participants showed overwhelming support for both potential usage of the dataset (cf. Figure 7.6) and the need for a search engine (cf. Figure 7.7).

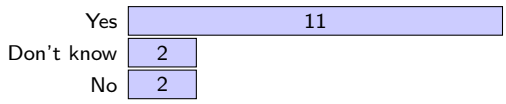


Figure 7.6: Responses to "We collected 9,117 open-source models from GitHub. Intuitively, do you think this collection can provide you with suitable Simulink models for your research?"

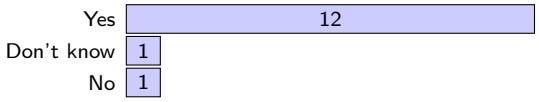


Figure 7.7: Responses to "Would you use ScoutSL for your research, in the future?"

7.5 Tool Architecture

Figure 7.8 illustrates ScoutSL's architecture, which consists of two main components: An (offline) mining component and an (online) web application component. The miner retrieves Simulink projects from the repository hosting sites and stores project, model, and commit metrics in a SQLite database.

An intermediate component queries the SQLite database, computes derived attributes such as a model's code generation capability, and calculates a "relevance" project score. A subset[1] of the primary and derived attributes are then stored in a cloud-hosted NoSQL database, as NoSQL databases typically have flexible data models and scale horizontally. The online web interface of ScoutSL facilitates user searches for Simulink projects, allowing filtering based on various attributes.

7.6 Mining Component

To mine from GitHub and MATLAB Central we use the existing SLNET [240] and EvoSL [194] infrastructure. Unlike SLNET or EvoSL, our focus is primarily on curating a comprehensive database of publicly available Simulink projects, and thus we do not prioritize the analysis of license files as the goal is to allow users to sample

[1]Due to unclear project licenses not all SQLite data are exposed.

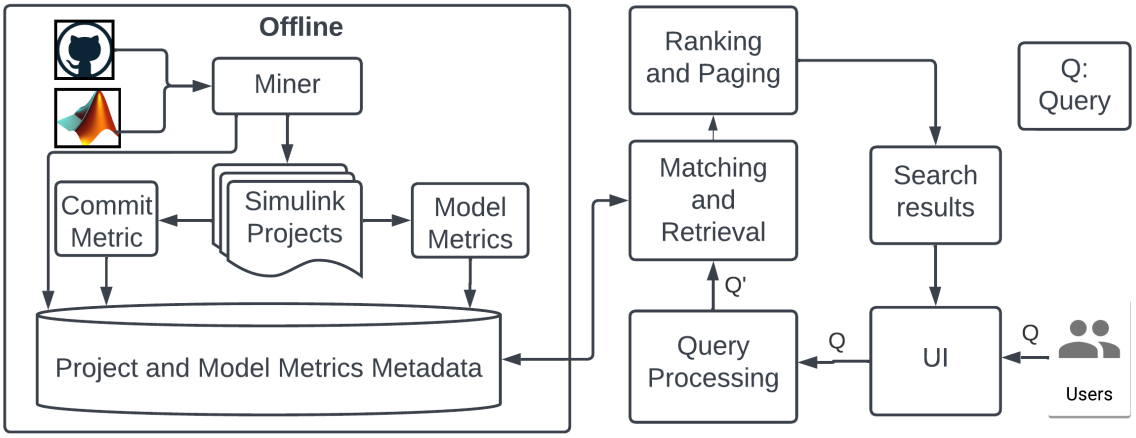


Figure 7.8: Architecture of the ScoutSL tool.

from all available open-source Simulink models. Our search yielded 18k projects comprising 109k Simulink models having 15M+ blocks ($\geq 15\times$ SLNET).

7.6.1 GitHub

Our extension of SLNET-Miner efficiently manages GitHub's API rate limits. Initially, we query for projects created within a specific time frame, such as *"q=simulink&created:2008-01-01..2009-01-01"*. To exhaustively search for projects using the GitHub API, we employ a divide-and-conquer strategy when the query returns 1k results. We split the time interval in half until the number of results returned is less than 1k.

From the query results we then iteratively download each project and check if it contains a Simulink model by checking for files with MDL or SLX extensions. We extract 80 attributes [243] from the projects' metadata, commits, issues, and pull requests. We only include each GitHub project's commits from its default-branch. Over a one-month period we thereby downloaded some 14k Simulink projects. To

mitigate redundancy we currently do not collect metrics from forked projects (but plan to add this in the future).

7.6.2 MATLAB Central

As SLNET we parse MATLAB Central's RSS feed (which does not impose any parsing restrictions), but the feed does not offer a structured method for downloading projects. We extended SLNET-Miner to enhance its heuristic for constructing download links for MATLAB Central projects. Despite these improvements, 7.9k of 46k MATLAB Central projects remained inaccessible for automated download.

While the GitHub API exposes a project's license name, MATLAB Central projects are bundled with license files that require further analysis. To automate this process, we utilized an open-source library employed by GitHub [245]. Mining MATLAB Central took about two and a half days and yielded 4.2k projects with 18 attributes [243].

7.6.3 Model Metrics

To facilitate searching based on Simulink model metrics, we extended the existing SLNET-Metrics tool to add more metrics, including the presence of a TargetLink blocks, toolbox dependencies, and system target files. Responding to survey responses (which highlighted researchers' interest in filtering models based on block categories), we further enhanced the tool to support the categorization of block types into non-overlapping categories, as employed in our recent work [194]. We analyzed models on MATLAB R2022b and collected 39 model metrics [243] overall.

7.7 User Interface

ScoutSL has simple and advanced search. The latter offers three distinct user interfaces: Simulink model search, repository search, and commit search (catering to various dataset research requirements, including model evolution studies and subject models for tool evaluations).

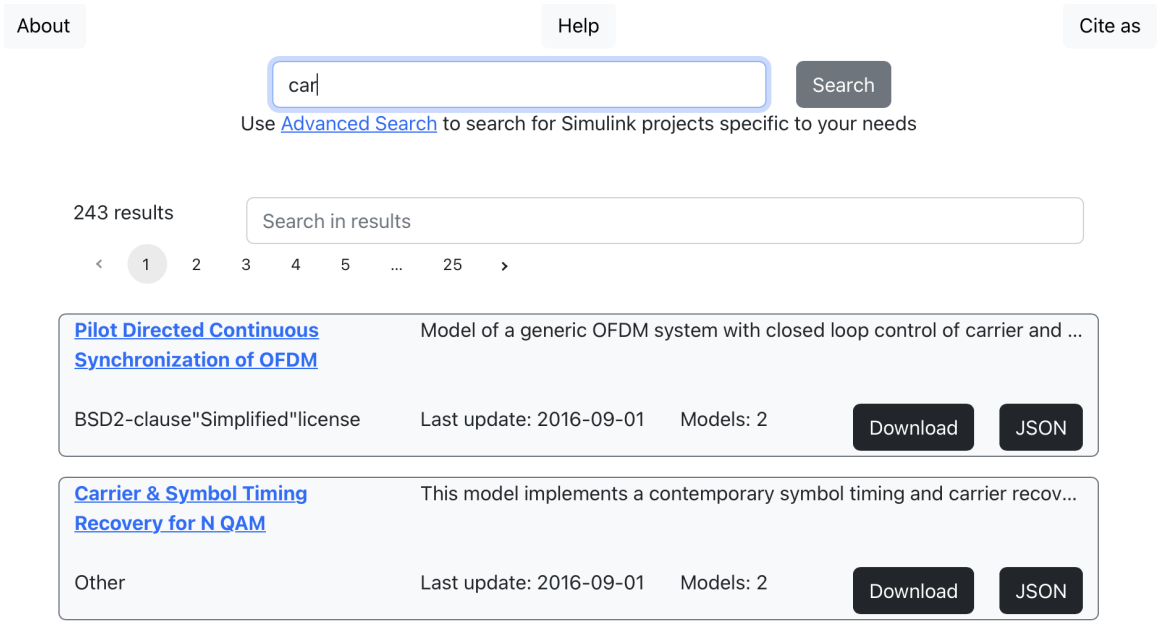


Figure 7.9: Example simple Simulink project search for "car".

7.7.1 Simple Search

In the simple search users enter a text-based query, which ScoutSL matches with the project descriptions in the database. An example search of "turbine" produces 50+ project results. ScoutSL then sorts the results in descending order based on a ranking score. To compute the ranking score, we adapted a strategy inspired by previous work [246] that aimed to classify engineered and toy projects. We selected the common Table 7.1 attributes shared by GitHub and MATLAB Central projects

and scored them based on survey responses. To prioritize Git-based attributes highlighted by the survey, we assigned lower weight scores to model revision and model contributors, while MATLAB Central projects received zero for these attributes.

Table 7.1: Project scoring scheme; CC = cyclomatic complexity.

Attribute	Survey	Weight	Scoring Scheme
Blocks	Size (12)	15	Continuous
Block types	Property (11)	15	Continuous
Code generation	Property (11)	15	Discrete (0,1)
Test harness	Property (11)	15	Discrete (0,1)
Documentation	Property (11)	15	Discrete (0,1)
License	License type (7)	10	Discrete (0,1)
CC	Complexity (5)	5	Continuous
Model revision	Git (2)	2	Continuous
Model contributors	Git (2)	2	Continuous
Toolbox	Add-ons (1)	1	Discrete (0,1)

We scored each attribute either via a binary (0 or 1) or a continuous scheme. For the latter we considered the distribution of data attributes, filtered out outliers, and normalized the scores to between 0 and 1. The final project score was calculated by summing the weighted Table 7.1 scores. While the scoring scheme is based on our survey responses, we do not make claims regarding the projects' engineering quality [246]. Evaluating and improving the scoring scheme is future work.

7.7.2 Advanced Search: Simulink Model

To cater to the goal of facilitating Simulink model sampling, Figure 7.10 illustrates ScoutSL's model search UI. The page highlights the specific model metrics that users expressed interest in, as determined through our survey. Users can input numeric values or select attribute options from drop-down menus to refine their search

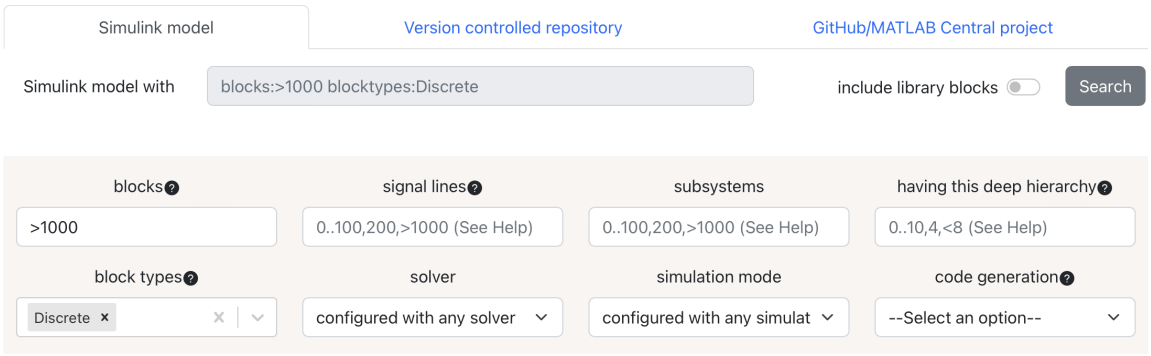


Figure 7.10: Example search for Simulink models that contain over 1k blocks, including some discrete blocks.

criteria. For example, they can search for Simulink models with over 1k blocks having discrete blocks. This search query generates a list of over 650 projects as search results.

7.7.3 Advanced Search: Simulink GitHub Repository

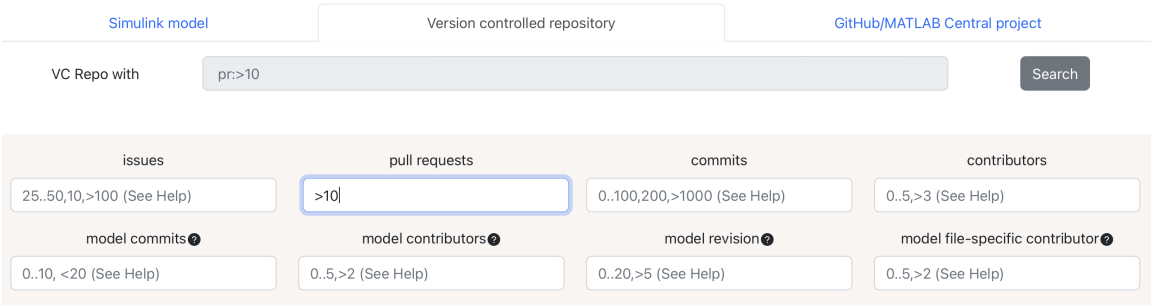


Figure 7.11: Example search for Simulink GitHub repositories that have over 10 pull requests.

In order to support studies on model evolution and changes, ScoutSL incorporates GitHub-based selection criteria focused on version control and project management, as seen in Figure 7.11. Specifically, users can employ search criteria such as the number of project issues, pull requests, commits, and contributors. With our

emphasis on Simulink models, users can also search for model-specific commits and contributions, which are subsets of project commits and contributors.

Additionally, ScoutSL enables users to filter projects based on a specific number of model revisions or model files with a certain number of authors. These model-specific search criteria are a unique ScoutSL feature not available on other online web-based tool. While other tools may offer project-level commit information and allow to search for commits per project [239], ScoutSL offers to search over the entire project database. As an example, researchers investigating model development can efficiently identify relevant projects by using a search query that filters for those with a significant number of model revisions. By using a search query for projects with more than 10 model revisions, ScoutSL yields over 500 relevant projects.

7.7.4 Advanced Search: Simulink Project

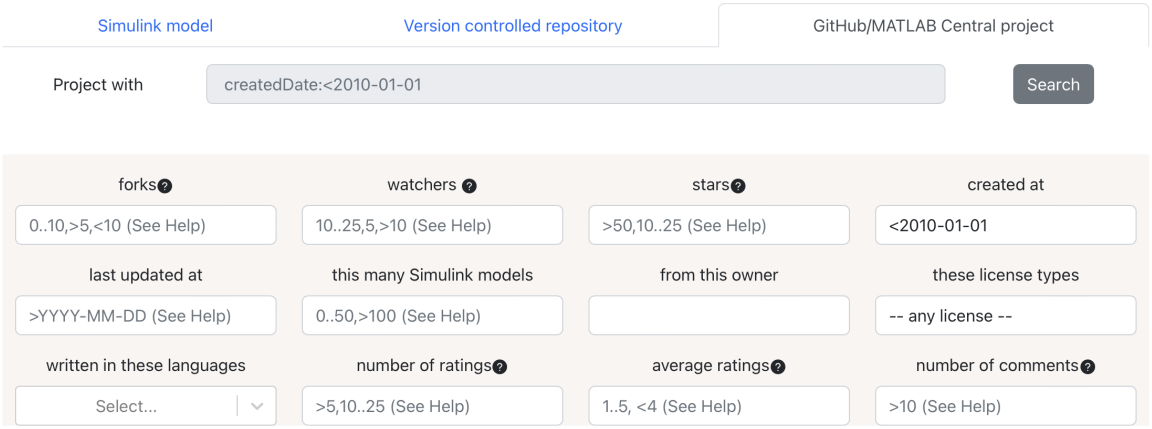


Figure 7.12: Example search for pre-2010 Simulink projects.

A commonly employed strategy in mining software repositories for exploratory studies is to sample projects based on popularity metrics. ScoutSL provides the capability to filter projects based on such metrics, as depicted in Figure 12. Additionally,

users are able to query projects created within specific date ranges. Another feature we offer is the ability to filter projects based on license type, as it was identified as a requested feature in the survey responses. Since a Simulink project is often accompanied by complementary scripts written in other programming languages, users can also perform searches involving such criteria. For instance, researchers interested in studying projects Simulink projects with JAVA and C code can use ScoutSL to get 250+ relevant projects. While the most up-to-date project attributes may be available through the primary hosting sites, our database exclusively comprises Simulink projects, which are not easily sampled using existing tools or the primary hosting sites from which we mine our projects.

7.8 Related Work

While several tools have been developed to facilitate sampling of open-source projects from platforms like GitHub, they primarily focus on textual programming languages [247, 248]. To obtain model artifacts, a web-based search tool, ModelMine [239], queries GitHub API to narrow down results with file extension. However, the tool mistakenly identifies ".simulink" as a Simulink model file extension and is inherently limited by GitHub API's 1k results per request limit.

A recent study examining forums of modeling tools including MATLAB/Simulink highlighted the potential benefits of model repositories, particularly for novice users who may encounter difficulties when attempting to model something specific [234]. The study emphasizes the importance of establishing and maintaining a diverse repository of example models. To that end, MAR [238] is a web-based search engine that maintains metamodels for various types of models, including UML models. For Simulink, the tool analyzes the pre-curated corpus to extract their metamodel using a

third-party tool. As such, using MAR requires knowledge of modelling languages like EMF to get relevant models, and the search space is limited to 200 Simulink models.

7.9 Conclusions and Future Work

ScoutSL (`http://scoutsl.net`) is the first search engine geared towards Simulink users' needs. ScoutSL allows searching over 18k Simulink projects containing over 100k Simulink models.

Future works include extension of mining tool to enlarge and augment the dataset with new primary as well as derived project/model attributes such as project domain, Simulink model version. We intend to improve the search engine performance and conduct a thorough evaluation.

CHAPTER 8

Conclusions

In summary, this dissertation proposes novel approaches to random Simulink model generation and presents the largest corpora of Simulink models and projects. DeepFuzzSL proved applicability of language models to generate random Simulink models and SLGPT leveraged recent advancements of natural language processing technique to limit the size of training samples required to learn a language models while improving the fidelity of the generated models to realistic Simulink model. SLNET and EvoSL addressed the need for easily accessible third-party Simulink models and projects. Our search engine, ScoutSL further lowered the barrier for sampling Simulink models and projects from open-source domain. Much of the work in this dissertation is aimed at promoting open science and encouraging research on the Simulink models by lowering the barrier to entry by providing large datasets. Given all the aspects of the research presented in this dissertation are publicly available, we hope that future work can build upon our work by enlarging the dataset, make further improvement on random model generation and use the dataset themselves to explore new and untapped research avenue with Simulink models.

REFERENCES

[1] G. J. Raju, "1.5mw wind generation plant," https://www.mathworks.com/matlabcentral/fileexchange/73469-1-5mw-wind-generation-plant, Dec. 2019, accessed Jan 2022.

[2] S. L. Shrestha, S. A. Chowdhury, and C. Csallner, "SLNET: A Redistributable Corpus of 3rd-party Simulink Models," Mar. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.5259648

[3] MathWorks Inc, "Compare and merge Simulink models containing Stateflow," 2023, accessed Mar 2022. [Online]. Available: https://www.mathworks.com/help/simulink/slref/compare-and-merge-simulink-models-containing-stateflow.html

[4] M. Jaskolka, V. Pantelic, A. Wassyng, M. Lawford, and R. Paige, "Repository mining for changes in Simulink models," in *MODELS*. IEEE, 2021, pp. 46–57.

[5] M. Jaskolka and Gor-Marks, "McSCert/Model-Comparison-Utility: Model Comparison Utility (Version 1.4)," Apr. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6410073

[6] A. Lima, L. Rossi, and M. Musolesi, "Coding together at scale: GitHub as a collaborative social network," in *Proc. 8th International Conference on Weblogs and Social Media (ICWSM)*. AAAI, June 2014.

[7] M. Y. Allaho and W. Lee, "Trends and behavior of developers in open collaborative software projects," in *2014 International Conference on Behavioral, Economic, and Socio-Cultural Computing, BESC 2014, Shanghai,*

China, October 30 - November 1, 2014. IEEE, 2014, pp. 96–102. [Online]. Available: https://doi.org/10.1109/BESC.2014.7059515

[8] W. Ma, L. Chen, Y. Zhou, and B. Xu, "What are the dominant projects in the GitHub Python ecosystem?" in *Proc. 3rd International Conference on Trustworthy Systems and their Applications (TSA)*. IEEE, Sept. 2016, pp. 87–95.

[9] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. Germán, and D. E. Damian, "An in-depth study of the promises and perils of mining GitHub," *Empir. Softw. Eng.*, vol. 21, no. 5, pp. 2035–2071, 2016.

[10] MathWorks, "Matlab and simulink help land unpiloted boeing spacecraft," 2023, november 2023. [Online]. Available: https://www.mathworks.com/company/user_stories/matlab-and-simulink-help-land-unpiloted-boeing-spacecraft.html

[11] C. Gadda and A. Simpso, "Using model-based design to build the tesla roadster," 2023, november 2023. [Online]. Available: https://www.mathworks.com/company/newsletters/articles/using-model-based-design-to-build-the-tesla-roadster.html

[12] A. A. Wright and A. Cashion, "Developing a transmitter for high-temperature, high-speed geothermal data transfer," 2023, november 2023. [Online]. Available: https://www.mathworks.com/company/newsletters/articles/developing-a-transmitter-for-high-temperature-high-speed-geothermal-data-transfer.html?s_tid=srchtitle

[13] MathWorks, "Mathworks product release schedule," 2023, october 2023. [Online]. Available: https://www.mathworks.com/products/new_products/release_model.html#:~:text=MathWorks%20follows%20a%20twice%2Dyearly,%2C%20when%20available%2C%20new%20products.

[14] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Trans. Software Eng.*, vol. 41, no. 5, pp. 507–525, 2015.

[15] A. Groce, G. J. Holzmann, and R. Joshi, "Randomized differential testing as a prelude to formal verification," in *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. IEEE Computer Society, 2007, pp. 621–631. [Online]. Available: https://doi.org/10.1109/ICSE.2007.68

[16] S. A. Chowdhury, S. Mohian, S. Mehra, S. Gawsane, T. T. Johnson, and C. Csallner, "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, May 2018, paper http://ranger.uta.edu/ csallner/papers/Chowdhury18SLforge.pdf.

[17] S. A. Chowdhury, T. T. Johnson, and C. Csallner, "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, Oct. 2016, paper http://ranger.uta.edu/ csallner/papers/-Chowdhury16CyFuzz.pdf, pp. 46–60.

[18] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998. [Online]. Available: http://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf

[19] MathWorks Inc, "Simulation phases in dynamic systems," https://www.mathworks.com/help/simulink/ug/simulating-dynamic-systems.html, accessed Jan 2020.

[20] MathWorks, "Interact with the acceleration modes programmatically," 2023, october 2023. [Online]. Available: https://www.mathworks.com/help/simulink/ug/interacting-with-the-acceleration-modes-programmatically.html

[21] ——, "Choosing a simulation mode," 2023, october 2023. [Online]. Available: https://www.mathworks.com/help/simulink/ug/choosing-a-simulation-mode.html

[22] S. A. Chowdhury, S. Mohian, S. Mehra, S. Gawsane, T. T. Johnson, and C. Csallner, "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)*, May 2018, pp. 981–992.

[23] X. Liu, X. Li, R. Prajapati, and D. Wu, "Deepfuzz: Automatic generation of syntax valid C programs for fuzz testing," in *Proc. 33rd AAAI Conference on Artificial Intelligence (AAAI)*. AAAI, Jan. 2019, pp. 1044–1051.

[24] H. Xu, Y. Wang, S. Fan, P. Xie, and A. Liu, "Dsmith: Compiler fuzzing through generative deep learning model with attention," in *2020 International Joint Conference on Neural Networks, IJCNN 2020*, 2020.

[25] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013.*, 2013, pp. 3111–3119.

[26] Y. Bengio, R. Ducharme, and P. Vincent, "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, 2001, pp. 932–938.

[27] S. L. Shrestha, S. A. Chowdhury, and C. Csallner, "DeepFuzzSL: Generating models with deep learning to find bugs in the Simulink toolchain,"

in *DeepTest 2020*, May 2020, paper http://ranger.uta.edu/ csallner/papers/Shrestha20DeepFuzzSL.pdf.

[28] S. L. Shrestha, "Automatic generation of Simulink models to find bugs in a cyber-physical system tool chain using deep learning," pp. 110–112, 2020. [Online]. Available: https://doi.org/10.1145/3377812.3382163

[29] S. L. Shrestha and C. Csallner, "SLGPT: Using transfer learning to directly generate Simulink model files and find bugs in the Simulink toolchain," in *EASE*. ACM, 2021, pp. 260–265. [Online]. Available: https://doi.org/10.1145/3463274.3463806

[30] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI blog*, 2019.

[31] S. A. Chowdhury, L. S. Varghese, S. Mohian, T. T. Johnson, and C. Csallner, "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, May 2018, pp. 45–48.

[32] S. L. Shrestha, S. A. Chowdhury, and C. Csallner, "SLNET: A Redistributable Corpus of 3rd-party Simulink models," in *MSR*. IEEE, May 2022, pp. 1–5.

[33] A. Boll, N. Vieregg, and T. Kehrer, "Replicability of experimental tool evaluations in model-based software and systems engineering with matlab/simulink," *Innovations in Systems and Software Engineering*, pp. 1–16, 2022.

[34] S. L. Shrestha, A. Boll, T. Kehrer, and C. Csallner, "ScoutSL: An open-source simulink search engine," in *MODELS-C*. IEEE, 2023, pp. 70–74, 2023 IEEE. Reprinted, with permission, from S. L. Shrestha, A. Boll, T. Kehrer and C. Csallner, "ScoutSL: An Open-Source Simulink Search Engine," 2023 ACM/IEEE International Conference on Model Driven Engineering Languages

and Systems Companion (MODELS-C), Västerås, Sweden, 2023, pp. 70-74, doi: 10.1109/MODELS-C59198.2023.00022.

[35] MathWorks Inc., "MATLAB & simulink," https://www.mathworks.com/products/simulink.html/, 2020, accessed Jan 2020.

[36] Y. Chen, A. Groce, C. Zhang, W. Wong, X. Z. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, 2013, pp. 197–208.

[37] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "An empirical comparison of compiler testing techniques," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 180–190.

[38] X. Liu, X. Li, R. Prajapati, and D. Wu, "Deepfuzz: Automatic generation of syntax valid C programs for fuzz testing," in *The Thirty-Third AAAI Conference on Artificial Intelligence*, 2019, pp. 1044–1051.

[39] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler fuzzing through deep learning," in *ISSTA 2018*, 2018.

[40] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[41] N. Pham, G. Kruszewski, and G. Boleda, "Convolutional neural network language models," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016*, 2016, pp. 1153–1162.

[42] S. L. Shrestha, S. A. Chowdhury, and C. Csallner, "50417/deepfuzzsl: Deepfuzzsl first release," Mar. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3712482

[43] R. Puri, R. Kirby, N. Yakovenko, and B. Catanzaro, "Large scale language modeling: Converging on 40gb of text in four hours," in *30th International Symposium on Computer Architecture and High Performance Computing*, 2018, pp. 290–297.

[44] Z. C. Lipton, "A critical review of recurrent neural networks for sequence learning," *ArXiv*, vol. abs/1506.00019, 2015.

[45] M. Wolf and E. Feron, "What don't we know about CPS architectures?" in *Proceedings of the 52nd Annual Design Automation Conference*, 2015, pp. 80:1–80:4.

[46] MathWorks Inc, "Simulink block libraries documentation," https://www.mathworks.com/help/simulink/block-libraries.html, accessed Jan 2020.

[47] ——, "How accelerator model works documentation," https://www.mathworks.com/help/simulink/ug/how-the-acceleration-modes-work.htm, accessed Jan 2020.

[48] MathWorks Inc., "Simulink documentation," https://www.mathworks.com/help/simulink/, 2019, accessed Jan 2020.

[49] X. Liu, D. Cao, and K. Yu, "Binarized LSTM language model," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics(ACL): Human Language Technologies, Volume 1 (Long Papers)*. ACL, June 2018, pp. 2113–2121.

[50] A. Radford, R. Józefowicz, and I. Sutskever, "Learning to generate reviews and discovering sentiment," *arXiv preprint arXiv:1704.01444*, 2017. [Online]. Available: http://arxiv.org/abs/1704.01444

[51] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[52] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[53] TACC at The University of Texas at Austin. (2021) Texas advanced computing center - homepage. [Online]. Available: https://www.tacc.utexas.edu/

[54] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. 3rd International Conference on Learning Representations (ICLR)*, 2015.

[55] A. Holtzman, J. Buys, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," *CoRR*, 2019. [Online]. Available: http://arxiv.org/abs/1904.09751

[56] C. Csallner and Y. Smaragdakis, "JCrasher: An automatic robustness tester for Java," *Software—Practice & Experience*, vol. 34, no. 11, Sept. 2004.

[57] K. Dewey, J. Roesch, and B. Hardekopf, "Fuzzing the rust typechecker using CLP (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, 2015, pp. 482–493.

[58] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," *CoRR*, vol. abs/1608.01723, 2016. [Online]. Available: http://arxiv.org/abs/1608.01723

[59] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, 2011, pp. 283–294. [Online]. Available: https://doi.org/10.1145/1993498.1993532

[60] S. A. Chowdhury, S. L. Shrestha, T. T. Johnson, and C. Csallner, "SLEMI: Equivalence modulo input (EMI) based mutation of CPS models for finding compiler bugs in Simulink," in *ICSE*. ACM, May 2020, pp. 335–346.

[61] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: machine learning for input fuzzing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017*, 2017.

[62] Y. Chen, C. M. Poskitt, J. Sun, S. Adepu, and F. Zhang, "Learning-guided network fuzzing for testing cyber-physical system defences," in *Proc. 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Nov. 2019, pp. 962–973.

[63] S. Raj, S. K. Jha, A. Ramanathan, and L. L. Pullum, "Testing autonomous cyber-physical systems using fuzzing features from convolutional neural networks: work-in-progress," in *Proc. 13th ACM International Conference on Embedded Software (EMSOFT) Companion*, 2017, pp. 1:1–1:2.

[64] A. Bezemskij, G. Loukas, D. Gan, and R. J. Anthony, "Detecting cyber-physical threats in an autonomous robotic vehicle using bayesian networks," in *2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, 2017.

[65] B. Liu, Lucia, S. Nejati, and L. C. Briand, "Improving fault localization for simulink models using search-based testing and prediction models," in *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, 2017, pp. 359–370.

[66] M. Kravchik and A. Shabtai, "Detecting cyber attacks in industrial control systems using convolutional neural networks," in *Proceedings of the 2018 Workshop on Cyber-Physical Systems Security and PrivaCy, CPS-SPC@CCS 2018, Toronto, ON, Canada, October 19, 2018*, 2018, pp. 72–83.

[67] X. Zheng, C. Julien, M. Kim, and S. Khurshid, "Perceptions on the state of the art in verification and validation in cyber-physical systems," *IEEE Systems Journal*, vol. 11, no. 4, pp. 2614–2627, 2017. [Online]. Available: https://doi.org/10.1109/JSYST.2015.2496293

[68] S. L. Shrestha, "50417/slgpt," May 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4734223

[69] MathWorks Inc, "MATLAB & Simulink," 2023, accessed June 2023. [Online]. Available: https://www.mathworks.com/products/simulink.html/

[70] M. Inc., "Simulink® 7 reference," pp. Pg. 9–2 – Pg.9–10, 9 2007.

[71] MathWorks Inc. (2021) Block libraries. [Online]. Available: https://www.mathworks.com/help/simulink/block-libraries.html

[72] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, 2009.

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

[74] J. Howard and S. Ruder, "Universal language model fine-tuning for text classification," in *ACL 2018*. ACL, July 2018, pp. 328–339.

[75] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *30th Annual Conference on Neural Information Processing Systems*, Sept. 2017.

[76] A. Holtzman, J. Buys, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," *CoRR*, vol. abs/1904.09751, 2019.

[77] S. Fakhoury, V. Arnaoudova, C. Noiseux, F. Khomh, and G. Antoniol, "Keep it simple: Is deep learning good for linguistic smell detection?" in *SANER'18*. IEEE, May 2018, pp. 602–611.

[78] R. Robbes and A. Janes, "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, May 2019.

[79] V. Raychev, M. T. Vechev, and E. Yahav, "Code completion with statistical language models," in *PLDI 2014*, 2014, pp. 419–428.

[80] M. Hammad, Ö. Babur, H. A. Basit, and M. van den Brand, "Deepclone: Modeling clones to generate code predictions," in *ICSR 2020*, Dec. 2020.

[81] Q. Chen and M. Zhou, "A neural framework for retrieval and summarization of source code," in *ASE 2018*, 2018.

[82] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler fuzzing through deep learning," in *ISSTA 2018*, 2018.

[83] J. Cruz-Benito, S. Vishwakarma, F. Martin-Fernandez, and I. Faro, "Automated source code generation and auto-completion using deep learning: Comparing and discussing current language model-related approaches," *AI*, vol. 2, no. 1, pp. 1–16, 2021. [Online]. Available: https://www.mdpi.com/2673-2688/2/1/1

[84] Y. Hussain, Z. Huang, Y. Zhou, and S. Wang, "Deep transfer learning for source code modeling," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 30, no. 5, 2020.

[85] S. L. Shrestha, S. A. Chowdhury, and C. Csallner, "SLNET: A redistributable corpus of 3rd-party Simulink models," in *Proc. 19th IEEE/ACM International Conference on Mining Software Repositories (MSR)*. IEEE, May 2022, pp. 237–241.

[86] M. Stephan and J. R. Cordy, "Identification of simulink model antipattern instances using model clone detection," in *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS*

2015, 2015, pp. 276–285. [Online]. Available: https://doi.org/10.1109/MODELS.2015.7338258

[87] M. Stephan, M. H. Alalfi, J. R. Cordy, and A. Stevenson, "Evolution of model clones in simulink," in *Proc. Workshop on Models and Evolution*, 2013, pp. 40–49.

[88] M. Wolf and E. Feron, "What don't we know about CPS architectures?" in *Proc. 52nd Annual Design Automation Conference (DAC)*, June 2015, pp. 80:1–80:4.

[89] A. Boll, F. Brokhausen, T. Amorim, T. Kehrer, and A. Vogelsang, "Characteristics, potentials, and limitations of open-source simulink projects for empirical research," *Softw. Syst. Model.*, vol. 20, no. 6, pp. 2111–2130, 2021.

[90] E. J. Rapos and J. R. Cordy, "Simevo: A toolset for simulink test evolution & maintenance," in *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*. IEEE Computer Society, 2018, pp. 410–415. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ICST.2018.00049

[91] S. L. Shrestha, "SLNET-Miner," 2022, November 2022. [Online]. Available: https://github.com/50417/SLNet_Miner

[92] ——, "50417/SLNET_Metrics: SLNET_Metrics MSR Release," Mar. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6336048

[93] GitHub Inc, "GitHub," 2023, accessed June 2023. [Online]. Available: https://github.com

[94] MathWorks Inc, "Matlab central," 2021, accessed Nov 2021. [Online]. Available: https://www.mathworks.com/matlabcentral/fileexchange/

[95] GitLab, "Gitlab," 2023, accessed May 2021. [Online]. Available: https://gitlab.com

[96] Slashdot Media, "Sourceforge," 2023, accessed May 2021. [Online]. Available: https://sourceforge.net/directory/?q=simulink

[97] Vincent Jacques, "Pygithub," 2023, accessed June 2023. [Online]. Available: https://pygithub.readthedocs.io/en/latest/introduction.html

[98] GitHub Inc, "Github developer," 2020, february 2020. [Online]. Available: https://developer.github.com/v3/search/

[99] MathWorks Inc, "Mathworks rss," 2023, february 2023. [Online]. Available: https://www.mathworks.com/company/rss.html

[100] S. L. Shrestha, "Matlab Simulink installation," 2021, accessed Nov 2021. [Online]. Available: https://github.com/50417/SLNET_Metrics/wiki/MATLAB-Simulink-Installation

[101] G. Rouleau, "How many blocks are in that model?" 2023, accessed June 2023. [Online]. Available: https://blogs.mathworks.com/simulink/2009/08/11/how-many-blocks-are-in-that-model

[102] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Appl. Soft Comput.*, vol. 27, pp. 504–518, 2015. [Online]. Available: https://doi.org/10.1016/j.asoc.2014.11.023

[103] M. Zanoni, F. A. Fontana, and F. Stella, "On applying machine learning techniques for design pattern detection," *J. Syst. Softw.*, vol. 103, pp. 102–117, 2015. [Online]. Available: https://doi.org/10.1016/j.jss.2015.01.037

[104] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proc. ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, Aug. 2009, pp. 213–222.

[105] B. Adhikari, "Intelligent simulink modeling assistance via model clones and machine learning," p. 237, 2021.

[106] Y. Chen, C. M. Poskitt, J. Sun, S. Adepu, and F. Zhang, "Learning-guided network fuzzing for testing cyber-physical system defences," in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 962–973. [Online]. Available: https://doi.org/10.1109/ASE.2019.00093

[107] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J. Girard, and S. Teuchert, "Clone detection in automotive model-based development," in *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, 2008, pp. 603–612. [Online]. Available: https://doi.org/10.1145/1368088.1368172

[108] V. Pantelic, S. M. Postma, M. Lawford, M. Jaskolka, B. Mackenzie, A. Korobkine, M. Bender, J. Ong, G. Marks, and A. Wassyng, "Software engineering practices and simulink: bridging the gap," *STTT*, vol. 20, no. 1, pp. 95–117, 2018. [Online]. Available: https://doi.org/10.1007/s10009-017-0450-9

[109] R. Reicherdt and S. Glesner, "Slicing MATLAB simulink models," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, 2012, pp. 551–561. [Online]. Available: https://doi.org/10.1109/ICSE.2012.6227161

[110] T. Gerlitz, Q. M. Tran, and C. Dziobek, "Detection and handling of model smells for matlab/simulink models," in *Proc. International Workshop on Modelling in Automotive Software Engineering*, vol. 1487. CEUR-WS.org, Sept. 2015, pp. 13–22.

[111] A. Boll and T. Kehrer, "On the replicability of experimental tool evaluations in model-based development - lessons learnt from a systematic literature review focusing on matlab/simulink," in *Proc. 1st International Conference on Systems Modelling and Management (ICSMM)*. Springer, June 2020, pp. 111–130.

[112] Y. Dajsuren, M. G. J. van den Brand, A. Serebrenik, and S. A. Roubtsov, "Simulink models are also software: modularity assessment," in *Proc. 9th International ACM SIGSOFT conference on Quality of Software Architectures (QoSA)*, June 2013, pp. 99–106.

[113] M. Olszewska, Y. Dajsuren, H. Altinger, A. Serebrenik, M. A. Waldén, and M. G. J. van den Brand, "Tailoring complexity metrics for Simulink models," in *Proc. 10th European Conference on Software Architecture Workshops*, Nov. 2016, p. 5.

[114] E. A. Antonio, F. Ferrari, G. A. d. P. Caurin, and S. C. Fabbri, "A set of metrics for characterizing simulink model comprehension," *Journal of Computer Science and Technology*, vol. 14, no. 02, pp. 88–94, 2014.

[115] J. Schroeder, C. Berger, T. Herpel, and M. Staron, "Comparing the applicability of complexity measurements for simulink models during integration testing - an industrial case study," in *Proc. 2nd IEEE/ACM International Workshop on Software Architecture and Metrics (SAM)*. IEEE, May 2015, pp. 35–40.

[116] J. Cabot and M. Gogolla, "Object constraint language (OCL): A definitive guide," in *Proc. 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems*. Springer, 2012, pp. 58–90.

[117] J. Noten, J. Mengerink, and A. Serebrenik, "A data set of OCL expressions on github," in *Proc. 14th International Conference on Mining Software Repositories (MSR)*. IEEE, May 2017, pp. 531–534.

[118] J. G. M. Mengerink, J. Noten, R. R. H. Schiffelers, M. G. J. van den Brand, and A. Serebrenik, "A case of industrial vs. open-source OCL: not so different after all," in *Proc. MODELS Posters*, vol. 2019. CEUR-WS.org, Sept. 2017, pp. 472–474.

[119] S. L. Shrestha, S. A. Chowdhury, and C. Csallner, "Replicability study: Corpora for understanding simulink models & projects," in *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2023, pp. 130–141, 2023 IEEE. Reprinted, with permission, from S. L. Shrestha, S. A. Chowdhury and C. Csallner, "Replicability Study: Corpora For Understanding Simulink Models & Projects" in 2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), New Orleans, LA, USA, 2023, pp. 130-141, doi: 10.1109/ESEM56168.2023.10304867. [Online]. Available: https://doi.org/10.1109/ESEM56168.2023.10304867

[120] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann, "Effective test suites for mixed discrete-continuous stateflow controllers," in *ESEC/FSE*. ACM, Aug. 2015, pp. 84–95.

[121] A. C. Rao, A. Raouf, G. Dhadyalla, and V. Pasupuleti, "Mutation testing based evaluation of formal verification tools," in *International Conference on Dependable Systems and Their Applications, DSA 2017, Beijing, China, October 31 - November 2, 2017*. IEEE, 2017, pp. 1–7. [Online]. Available: https://doi.org/10.1109/DSA.2017.10

[122] Z. Jiang, X. Wu, Z. Dong, and M. Mu, "Optimal test case generation for Simulink models using slicing," in *Proc. IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, July 2017, pp. 363–369.

[123] V. Bertram, S. Maoz, J. O. Ringert, B. Rumpe, and M. von Wenckstern, "Component and connector views in practice: An experience report," in *MODELS*. IEEE Computer Society, September 2017, pp. 167–177.

[124] D. T. Campbell and J. C. Stanley, *Experimental and Quasi-Experimental Designs for Research*. Houghton Mifflin Company, 1963, p. 3.

[125] S. L. Shrestha, S. A. Chowdhury, and C. Csallner, "Replicability study: Corpora for understanding simulink models & projects (analysis data) and slnet-evol dataset," July 2023. [Online]. Available: https://figshare.com/articles/dataset/Replicability_Study_Corpora_For_Understanding_Simulink_Models_Projects/22064969

[126] S. L. Shrestha, "50417/SLReplicationTool: Replicability Study: Corpora For Understanding Simulink Models & Projects," July 2023. [Online]. Available: https://doi.org/10.5281/zenodo.8111687

[127] MathWorks Inc, "Choose among types of model components," 2023, accessed April 2023. [Online]. Available: https://www.mathworks.com/help/simulink/ug/types-of-model-components.html

[128] ——, "Linked blocks," 2023, June 2023. [Online]. Available: https://www.mathworks.com/help/simulink/ug/creating-and-working-with-linked-blocks.html

[129] ——, "Custom libraries," 2023, June 2023. [Online]. Available: https://www.mathworks.com/help/simulink/libraries.html

[130] ——, "How accelerator model works documentation," 2021, accessed Jan 2022. [Online]. Available: https://www.mathworks.com/help/simulink/ug/how-the-acceleration-modes-work.htm

[131] ——, "Products and services," 2022, accessed Nov 2022. [Online]. Available: https://www.mathworks.com/products.html

[132] MathWorks Advisory Board (MAB), "Control algorithm modeling guidelines using MATLAB, Simulink, and Stateflow," MathWorks Inc, Tech. Rep.

Version 5.0, 2020. [Online]. Available: https://www.mathworks.com/solutions/mab-guidelines.html

[133] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. 2, no. 4, pp. 308–320, Dec. 1976.

[134] O. Levy and D. G. Feitelson, "Understanding large-scale software systems - structure and flows," *Empir. Softw. Eng.*, vol. 26, no. 3, May 2021.

[135] D. G. Feitelson, "Considerations and pitfalls for reducing threats to the validity of controlled experiments on code comprehension," *Empir. Softw. Eng.*, vol. 27, no. 6, Nov. 2022.

[136] S. A. Chowdhury, R. Holmes, A. Zaidman, and R. Kazman, "Revisiting the debate: Are code metrics useful for measuring maintenance effort?" *Empir. Softw. Eng.*, vol. 27, no. 6, Nov. 2022.

[137] G. A. Campbell, "Cognitive complexity: An overview and evaluation," in *TechDebt*. ACM, May 2018, pp. 57–58.

[138] M. M. Barón, M. Wyrich, and S. Wagner, "An empirical validation of cognitive complexity as a measure of source code understandability," in *ESEM*. ACM, Oct. 2020, pp. 5:1–5:12.

[139] S. Ajami, Y. Woodbridge, and D. G. Feitelson, "Syntax, predicates, idioms - what really affects code complexity?" *Empir. Softw. Eng.*, vol. 24, no. 1, pp. 287–328, Feb. 2019.

[140] G. Jay, J. E. Hale, R. K. Smith, D. P. Hale, N. A. Kraft, and C. Ward, "Cyclomatic complexity and lines of code: Empirical evidence of a stable linear relationship," *J. Softw. Eng. Appl.*, vol. 2, no. 3, pp. 137–143, 2009.

[141] D. Landman, A. Serebrenik, E. Bouwers, and J. J. Vinju, "Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods and C functions," *J. Softw. Eng. Appl.*, vol. 28, no. 7, pp. 589–618, July 2016.

[142] J. Schroeder, C. Berger, M. Staron, T. Herpel, and A. Knauss, "Unveiling anomalies and their impact on software quality in model-based automotive software revisions with software metrics and domain experts," in *ISSTA*. ACM, July 2016, pp. 154–164.

[143] MathWorks Inc, "Multiport switch," 2023, accessed April 2023. [Online]. Available: https://www.mathworks.com/help/simulink/slref/multiportswitch.html

[144] ——, "Cyclomatic complexity metric," 2023, accessed April 2023. [Online]. Available: https://www.mathworks.com/help/slcheck/ug/mathworks.metrics.cyclomaticcomplexity.html

[145] B. Balasubramaniam, H. Bagheri, S. Elbaum, and J. Bradley, "Investigating controller evolution and divergence through mining and mutation*," in *ICCPS*, 2020, pp. 151–161.

[146] W. Hu, T. Loeffler, and J. Wegener, "Quality model based on iso/iec 9126 for internal quality of matlab/simulink/stateflow models," in *ICIT*. IEEE, 2012, pp. 325–330.

[147] H. Altinger, S. Siegl, Y. Dajsuren, and F. Wotawa, "A novel industry grade dataset for fault prediction based on model-driven developed automotive embedded software," in *Proc. 12th IEEE/ACM Working Conference on Mining Software Repositories (MSR)*. IEEE, May 2015, pp. 494–497.

[148] J. Schroeder, C. Berger, T. Herpel, and M. Staron, "Comparing the applicability of complexity measurements for Simulink models during integration testing – an industrial case study," in *SAM*. IEEE, May 2015, pp. 35–40.

[149] M. A. A. Mamun, C. Berger, and J. Hansson, "Effects of measurements on correlations of software code metrics," *Empir. Softw. Eng.*, vol. 24,

no. 4, pp. 2764–2818, 2019. [Online]. Available: https://doi.org/10.1007/s10664-019-09714-9

[150] A. Boll, T. Kehrer, A. Vogelsang, T. Amorim, and F. Brokhausen, "Characteristics, potentials, and limitations of open source Simulink projects for empirical research: Dataset," June 2021. [Online]. Available: https://doi.org/10.6084/m9.figshare.13636589.v2

[151] R. J. Wieringa and M. Daneva, "Six strategies for generalizing software engineering theories," *Science of Computer Programming*, vol. 101, pp. 136–152, 2015. [Online]. Available: https://doi.org/10.1016/j.scico.2014.11.013

[152] K. Stol and B. Fitzgerald, "The ABC of software engineering research," *ACM Transactions on Software Engineering and Methodology*, vol. 27, no. 3, pp. 11:1–11:51, Sept. 2018.

[153] Shafiul Azam Chowdhury, "Home," 2022, accessed Nov 2022. [Online]. Available: https://github.com/corpussimulink/corpus/wiki

[154] MathWorks Inc, "Simulink Check," 2023, accessed June 2023. [Online]. Available: https://www.mathworks.com/help/slcheck/

[155] Shafiul Azam Chowdhury, "ICSE 2018 Artifacts," 2022, accessed Nov 2022. [Online]. Available: https://github.com/verivital/slsf_randgen/wiki/ICSE-2018-Artifacts

[156] MathWorks Inc, "sldiagnostic," 2023, accessed June 2023. [Online]. Available: https://www.mathworks.com/help/simulink/slref/sldiagnostics.html

[157] D. Méndez, D. Graziotin, S. Wagner, and H. Seibold, "Open science in software engineering," in *Contemporary Empirical Methods in Software Engineering*. Springer, 2020, pp. 477–501.

[158] S. L. Shrestha, "Simulink Model Version," 2023, July 2023. [Online]. Available: https://github.com/50417/SLReplicationTool/blob/main/MatlabInstallation.md

[159] MathWorks Inc, "Enhanced calculation of cyclomatic complexity," 2023, accessed February 2023. [Online]. Available: https://www.mathworks.com/help/slcheck/release-notes.html

[160] Alan Hwang, "Video surveillance system design with simulink® and xilinx® fpgas," 2022, accessed Nov 2022. [Online]. Available: https://www.mathworks.com/matlabcentral/fileexchange/20160-video-surveillance-system-design-with-simulink-and-xilinx-fpgas

[161] MathWorks Inc, "bdislibrary," 2023, accessed April 2023. [Online]. Available: https://www.mathworks.com/help/simulink/slref/bdislibrary.html

[162] ——, "Simulink Test," 2023, accessed April 2023. [Online]. Available: https://www.mathworks.com/help/sltest/

[163] M. Jaskolka, S. Scott, V. Pantelic, A. Wassyng, and M. Lawford, "Applying modular decomposition in simulink," in *2020 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Coimbra, Portugal, October 12-15, 2020*. IEEE, 2020, pp. 31–36. [Online]. Available: https://doi.org/10.1109/ISSREW51248.2020.00033

[164] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann, "Test generation and test prioritization for Simulink models with dynamic behavior," *IEEE Trans. Software Eng.*, vol. 45, no. 9, pp. 919–944, 2019.

[165] A. Schlie, D. Wille, S. Schulze, L. Cleophas, and I. Schaefer, "Detecting variability in matlab/simulink models: An industry-inspired technique and its evaluation," in *Proceedings of the 21st International Systems and Software Product Line Conference, SPLC 2017, Volume A, Sevilla,*

Spain, September 25-29, 2017, 2017, pp. 215–224. [Online]. Available: https://doi.org/10.1145/3106195.3106225

[166] V. Pantelic, S. M. Postma, M. Lawford, M. Jaskolka, B. Mackenzie, A. Korobkine, M. Bender, J. Ong, G. Marks, and A. Wassyng, "Software engineering practices and simulink: Bridging the gap," *International Journal on Software Tools for Technology Transfer*, vol. 20, no. 1, pp. 95–117, 2018.

[167] B. Liu, Lucia, S. Nejati, and L. C. Briand, "Improving fault localization for Simulink models using search-based testing and prediction models," in *Proc. 24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb. 2017, pp. 359–370.

[168] S. A. Chowdhury, S. L. Shrestha, T. T. Johnson, and C. Csallner, "SLEMI: finding simulink compiler bugs through equivalence modulo input (EMI)," in *Proc. 42nd International Conference on Software Engineering (ICSE), Companion Volume*. ACM, May 2020, pp. 1–4. [Online]. Available: https://doi.org/10.1145/3377812.3382147

[169] B. Sánchez, A. Zolotas, H. H. Rodriguez, D. S. Kolovos, and R. F. Paige, "On-the-fly translation and execution of ocl-like queries on simulink models," in *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2019, Munich, Germany, September 15-20, 2019*, M. Kessentini, T. Yue, A. Pretschner, S. Voss, and L. Burgueño, Eds. IEEE, 2019, pp. 205–215. [Online]. Available: https://doi.org/10.1109/MODELS.2019.000-1

[170] MathWorks Inc, "Scope," 2023, accessed June 2023. [Online]. Available: https://www.mathworks.com/help/simulink/slref/scope.html

[171] ——, "Display," 2023, accessed June 2023. [Online]. Available: https://www.mathworks.com/help/simulink/slref/display.html

[172] MathWorks Inc., "To workspace," 2022, accessed Jan 2022. [Online]. Available: https://www.mathworks.com/help/simulink/slref/toworkspace.html

[173] MathWorks Inc, "From," 2021, accessed Nov 2021. [Online]. Available: https://www.mathworks.com/help/simulink/slref/from.html

[174] ——, "Goto," 2021, accessed Nov 2021. [Online]. Available: https://www.mathworks.com/help/simulink/slref/goto.html

[175] B. Adhikari, E. J. Rapos, and M. Stephan, "Simulink model transformation for backwards version compatibility," in *MODELS-C*. IEEE, Oct. 2021, pp. 427–436.

[176] S. L. Shrestha, S. A. Chowdhury, and C. Csallner, "DeepFuzzSL: Generating models with deep learning to find bugs in the Simulink toolchain," in *Proc. 2nd Workshop on Testing for Deep Learning and Deep Learning for Testing (DeepTest)*, May 2020, paper http://ranger.uta.edu/ csallner/papers/Shrestha20DeepFuzzSL.pdf.

[177] V. Pantelic, S. M. Postma, M. Lawford, A. Korobkine, B. Mackenzie, J. Ong, and M. Bender, "A toolset for Simulink: Improving software engineering practices in development with Simulink," in *MODELS*. SciTePress, February 2015, pp. 50–61.

[178] W. Ma, L. Chen, Y. Zhou, and B. Xu, "What are the dominant projects in the github python ecosystem?" in *TSA*. IEEE, September 2016, pp. 87–95.

[179] MathWorks Inc, "Model reference," 2022, accessed Nov 2022. [Online]. Available: https://www.mathworks.com/help/simulink/model-reference.html

[180] H. Zhang and H. B. K. Tan, "An empirical study of class sizes for large java systems," in *14th Asia-Pacific Software Engineering Conference (APSEC 2007), 5-7 December 2007, Nagoya, Japan*. IEEE Computer Society, 2007, pp. 230–237. [Online]. Available: https://doi.org/10.1109/APSEC.2007.20

[181] H. Zhang, H. B. K. Tan, and M. Marchesi, "The distribution of program sizes and its implications: An eclipse case study," in *1st International Symposium on Emerging Trends in Software Metrics*, 2009, pp. 1–10.

[182] S. W. Flint, J. Chauhan, and R. Dyer, "Pitfalls and guidelines for using time-based Git data," *Empirical Software Engineering*, vol. 27, no. 7, pp. 1–55, Dec. 2022.

[183] J. Krizan, L. Ertl, M. Bradac, M. Jasansky, and A. Andreev, "Automatic code generation from matlab/simulink for critical applications," in *IEEE 27th Canadian Conference on Electrical and Computer Engineering, CCECE 2014, Toronto, ON, Canada, May 4-7, 2014*. IEEE, 2014, pp. 1–6. [Online]. Available: https://doi.org/10.1109/CCECE.2014.6901058

[184] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann, "Automated test suite generation for time-continuous Smulink models," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 595–606. [Online]. Available: https://doi.org/10.1145/2884781.2884797

[185] M. M. R. Mozumdar, F. Gregoretti, L. Lavagno, L. Vanzago, and S. Olivieri, "A framework for modeling, simulation and automatic code generation of sensor network application," in *2008 5th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks*, 2008, pp. 515–522.

[186] H. Bourbouh, P.-L. Garoche, T. Loquen, É. Noulard, and C. Pagetti, "Cocosim, a code generation framework for control/command applications an overview of cocosim for multi-periodic discrete Simulink models," in *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*, 2020.

[187] MathWorks Inc, "Embedded coder," 2021, accessed Nov 2021. [Online]. Available: https://www.mathworks.com/products/embedded-coder.html

[188] dSpace, "Targetlink," 2022, accessed Jan 2022. [Online]. Available: https://www.dspace.com/en/inc/home/products/sw/pcgs/targetlink.cfm

[189] A. Boll, T. Kehrer, A. Vogelsang, T. Amorim, and F. Brokhausen, "Characteristics, potentials, and limitations of open source Simulink projects for empirical research: Dataset," Jan. 2021. [Online]. Available: https://doi.org/10.6084/m9.figshare.13636589.v1

[190] MathWorks Inc, "Treat as atomic unit," 2023, accessed April 2023. [Online]. Available: https://www.mathworks.com/help/simulink/slref/subsystem.html#brp1xt9-56

[191] ——, "Simulink Coder," 2023, accessed February 2023. [Online]. Available: https://www.mathworks.com/products/simulink-coder.html

[192] MathWorks Inc., "Target language compiler basics," 2022, accessed Jan 2022. [Online]. Available: https://www.mathworks.com/help/rtw/tlc/what-is-the-target-language-compiler.html

[193] MathWorks Inc, "Configure a system target file," 2021, accessed Nov 2021. [Online]. Available: https://www.mathworks.com/help/rtw/ug/select-a-target.html

[194] S. L. Shrestha, A. Boll, S. A. Chowdhury, T. Kehrer, and C. Csallner, "EvoSL: a large open-source corpus of changes in Simulink models & projects," in *MODELS*. IEEE, 2023, pp. 273–284, 2023 IEEE. Reprinted, with permission, from S. L. Shrestha, A. Boll, S. A. Chowdhury, T. Kehrer and C. Csallner, "EvoSL: A Large Open-Source Corpus of Changes in Simulink Models & Projects," 2023 ACM/IEEE International Conference on Model Driven Engineering Lan-

guages and Systems (MODELS), Västerås, Sweden, 2023, pp. 273-284, doi: 10.1109/MODELS58315.2023.00024.

[195] R. Jongeling, A. Cicchetti, F. Ciccozzi, and J. Carlson, "Co-evolution of Simulink models in a model-based product line," in *Proc. ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, E. Syriani, H. A. Sahraoui, J. de Lara, and S. Abrahão, Eds. ACM, Oct. 2020, pp. 263–273. [Online]. Available: https://doi.org/10.1145/3365438.3410989

[196] E. J. Rapos and J. R. Cordy, "Examining the co-evolution relationship between simulink models and their test cases," in *Proc. 8th International Workshop on Modeling in Software Engineering (MiSE)*. ACM, May 2016, pp. 34–40. [Online]. Available: https://doi.org/10.1145/2896982.2896983

[197] R. Heumüller, S. Nielebock, J. Krüger, and F. Ortmeier, "Publish or perish, but do not forget your software artifacts," *Empir. Softw. Eng.*, vol. 25, no. 6, pp. 4585–4616, 2020. [Online]. Available: https://doi.org/10.1007/s10664-020-09851-6

[198] S. L. Shrestha and C. Csallner, "SLGPT: Using transfer learning to directly generate Simulink model files and find bugs in the Simulink toolchain," in *Proc. 25th International Conference on Evaluation and Assessment in Software Engineering (EASE), Vision and Emerging Results Track*. ACM, 2021, pp. 260–265. [Online]. Available: https://doi.org/10.1145/3463274.3463806

[199] S. L. Shrestha, "EvoSL: A Large Open-Source Corpus of Changes in Simulink Models & Projects," Apr. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7806456

[200] S. L. Shrestha, A. Boll, S. A. Chowdhury, T. Kehrer, and C. Csallner, "50417/EvoSL-Tool: EvoSL: A Large Open-Source Corpus of Changes

in Simulink Models & Projects," July 2023. [Online]. Available: https://doi.org/10.5281/zenodo.8111019

[201] S. L. Shrestha, A. Boll, T. K. Shafiul Azam Chowdhury, and C. Csallner, "EvoSL: A large open-source corpus of changes in Simulink models & projects (analysis data)," Jul 2023. [Online]. Available: https://figshare.com/articles/dataset/EvoSL_A_Large_Open-Source_Corpus_of_Changes_in_Simulink_Models_Projects_Analysis_Data_/22298812

[202] MathWorks Inc, "Create block masks," 2022, accessed Nov 2022. [Online]. Available: https://www.mathworks.com/help/simulink/block-masks.html

[203] ——, "Set model configuration parameters for a model," 2022, accessed Nov 2022. [Online]. Available: https://www.mathworks.com/help/simulink/ug/configuration-parameters-dialog-box-overview.html

[204] ——, "Model comparison," Accessed December 2022, 2022. [Online]. Available: https://www.mathworks.com/help/simulink/model-comparison.html

[205] M. Stephan, M. H. Alalfi, and J. R. Cordy, "Towards a taxonomy for Simulink model mutations," in *Proc. 7th IEEE International Conference on Software Testing, Verification and Validation (ICST) Workshops*. IEEE, Mar. 2014, pp. 206–215.

[206] B. Adhikari, E. J. Rapos, and M. Stephan, "Simulink model transformation for backwards version compatibility," in *Proc. ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 2021, pp. 427–436.

[207] S. Guo, H. Jiang, Z. Xu, X. Li, Z. Ren, Z. Zhou, and R. Chen, "Detecting Simulink compiler bugs via controllable zombie blocks mutation," in *Proc. 30th ACM Symposium on the Foundations of Software Engineering (FSE)*. ACM, Nov. 2022, pp. 1061–1072.

[208] B. Adhikari, E. J. Rapos, and M. Stephan, "SimIMA: A virtual Simulink intelligent modeling assistant," *Software and Systems Modeling*, pp. 1619–1374, 2023. [Online]. Available: https://doi.org/10.1007/s10270-023-01093-6

[209] Software Heritage, "Faq–software heritage," 2023, accessed April 2023. [Online]. Available: https://www.softwareheritage.org/faq/#42_Can_I_clone_a_repository_using_SWH

[210] M. Z. Trujillo, L. Hébert-Dufresne, and J. P. Bagrow, "The penumbra of open source: Projects outside of centralized platforms are longer maintained, more academic and more collaborative," *EPJ Data Science*, vol. 11, no. 1, pp. 1–19, 2022.

[211] GitHub Inc, "Rate limits," 2023, accessed March 2023. [Online]. Available: https://docs.github.com/en/rest/overview/resources-in-the-rest-api?apiVersion=2022-11-28#rate-limiting

[212] D. Spadini, M. F. Aniche, and A. Bacchelli, "Pydriller: Python framework for mining software repositories," in *Proc. ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE*. ACM, Nov. 2018, pp. 908–911. [Online]. Available: https://doi.org/10.1145/3236024.3264598

[213] natedana, "How to copy a github repo without forking," 2023, march 2023. [Online]. Available: https://gist.github.com/natedana/cc71d496b611e70673cab5e8f5a78485

[214] Harvard Dataverse Project , "Harvard dataverse," 2023, april 2023. [Online]. Available: https://dataverse.harvard.edu/

[215] Digital Science, "Figshare," 2023, april 2023. [Online]. Available: https://figshare.com/

[216] IBM, "IBM Rational Change," 2023, april 2023. [Online]. Available: https://www.ibm.com/products/rational-change

[217] ——, "IBM Rational Synergy," 2023, april 2023. [Online]. Available: https://www.ibm.com/products/rational-synergy

[218] M. Jaskolka, V. Pantelic, A. Wassyng, R. Paige, and M. Lawford, "Repository mining for changes in Simulink and Stateflow models," *Software and Systems Modeling*, June 2023.

[219] MathWorks Inc, "DocBlock," 2023, april 2023. [Online]. Available: https://www.mathworks.com/help/simulink/slref/docblock.html

[220] ——, "Model Info," 2023, april 2023. [Online]. Available: https://www.mathworks.com/help/simulink/slref/modelinfo.html

[221] V. Pantelic, A. Schaap, A. Wassyng, V. Bandur, and M. Lawford, "Something is rotten in the state of documenting Simulink models," in *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2019, Prague, Czech Republic, February 20-22, 2019*. SciTePress, 2019, pp. 503–510. [Online]. Available: https://doi.org/10.5220/0007586005050512

[222] M. Kajko-Mattsson, "A survey of documentation practice within corrective maintenance," *Empir. Softw. Eng.*, vol. 10, no. 1, pp. 31–55, 2005. [Online]. Available: https://doi.org/10.1023/B:LIDA.0000048322.42751.ca

[223] E. Aghajani, C. Nagy, O. L. Vega-Márquez, M. Linares-Vásquez, L. Moreno, G. Bavota, and M. Lanza, "Software documentation issues unveiled," in *Proc. 41st IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1199–1210.

[224] A. Haber, C. Kolassa, P. Manhart, P. M. S. Nazari, B. Rumpe, and I. Schaefer, "First-class variability modeling in matlab/simulink," in *Proc.*

7th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS). ACM, Jan. 2013, pp. 4:1–4:8. [Online]. Available: https://doi.org/10.1145/2430502.2430508

[225] H. H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *J. Softw. Maintenance Res. Pract.*, vol. 19, no. 2, pp. 77–131, 2007. [Online]. Available: https://doi.org/10.1002/smr.344

[226] E. D. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The qualitas corpus: A curated collection of java code for empirical studies," in *17th Asia Pacific Software Engineering Conference, APSEC*, 2010, pp. 336–345.

[227] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *Proc. 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, May 2013, pp. 207–216. [Online]. Available: https://doi.org/10.1109/MSR.2013.6624029

[228] S. Romano, M. Caulo, M. Buompastore, L. Guerra, A. Mounsif, M. Telesca, M. T. Baldassarre, and G. Scanniello, "G-Repo: A tool to support MSR studies on GitHub," in *Proc. 28th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Mar. 2021, pp. 551–555.

[229] G. Gousios and D. Spinellis, "GHTorrent: Github's data from a firehose," in *Proc. 9th IEEE Working Conference of Mining Software Repositories (MSR)*. IEEE, June 2012, pp. 12–21.

[230] J. A. H. López and J. S. Cuadrado, "MAR: a structure-based search engine for models," in *Proc. ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, E. Syriani, H. A. Sahraoui,

J. de Lara, and S. Abrahão, Eds. ACM, Oct. 2020, pp. 57–67. [Online]. Available: https://doi.org/10.1145/3365438.3410947

[231] ——, "An efficient and scalable search engine for models," *Softw. Syst. Model.*, vol. 21, no. 5, pp. 1715–1737, 2022. [Online]. Available: https://doi.org/10.1007/s10270-021-00960-4

[232] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[233] E. D. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The Qualitas corpus: A curated collection of Java code for empirical studies," in *APSEC*. IEEE, Nov. 2010, pp. 336–345.

[234] C. Vendome, E. J. Rapos, and N. DiGennaro, "How do I model my system?: A qualitative study on the challenges that modelers experience," in *ICPC*. ACM, May 2022, pp. 648–659.

[235] S. L. Shrestha, "Harnessing large language models for simulink toolchain testing and developing diverse open-source corpora of simulink models for metric and evolution analysis," in *ISSTA*. ACM, 2023, pp. 1541–1545.

[236] GitHub Inc, "About," 2023, accessed in Mar 2023. [Online]. Available: https://github.com/about

[237] Thomas Dohmke, "100 million developers and counting," 2023, accessed on Mar 2023. [Online]. Available: https://github.blog/2023-01-25-100-million-developers-and-counting/

[238] J. A. H. López and J. S. Cuadrado, "An efficient and scalable search engine for models," *Softw. Syst. Model.*, vol. 21, pp. 1715–1737, 2022.

[239] S. M. Reza, O. Badreddin, and R. Khandoker, "ModelMine: A tool to facilitate mining models from open source repositories," in *MODELS*. ACM, Oct. 2020, pp. 9:1–9:5.

[240] S. L. Shrestha, S. A. Chowdhury, and C. Csallner, "SLNET: A redistributable corpus of 3rd-party Simulink models," in *MSR*, 2022, pp. 01–05. [Online]. Available: https://doi.org/10.1145/3524842.3528001

[241] S. A. Chowdhury, S. L. Shrestha, T. T. Johnson, and C. Csallner, "SLEMI: Equivalence modulo input (EMI) based mutation of CPS models for finding compiler bugs in Simulink," in *ICSE*. ACM, June 2020, pp. 335–346.

[242] S. L. Shrestha, A. Boll, T. Kehrer, and C. Csallner, "50417/ScoutSL: ScoutSL Simulink Search Engine," Aug. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.8266234

[243] ——, "ScoutSL Artifacts," Aug. 2023. [Online]. Available: https://figshare.com/articles/dataset/ScoutSL_Artifacts/23717763

[244] A. Boll, N. Vieregg, and T. Kehrer, "Replicability of experimental tool evaluations in model-based software and systems engineering with MATLAB/Simulink," *Innov. Syst. Softw. Eng.*, pp. 1–16, 2022.

[245] Licensee, "licensee," 2023, accessed in July 2023. [Online]. Available: https://github.com/licensee/licensee

[246] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating GitHub for engineered software projects," *Empir. Softw. Eng.*, vol. 22, no. 6, pp. 3219–3253, 2017.

[247] S. Romano, M. Caulo, M. Buompastore, L. Guerra, A. Mounsif, M. Telesca, M. T. Baldassarre, and G. Scanniello, "G-repo: A tool to support MSR studies on GitHub," in *SANER*. IEEE, Mar. 2021, pp. 551–555.

[248] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in GitHub for MSR studies," in *MSR*. IEEE, May 2021, pp. 560–564.