AUTOMATED TESTING OF A COMMERCIAL CYBER-PHYSICAL SYSTEM DEVELOPMENT TOOL CHAIN

by

SHAFIUL AZAM CHOWDHURY

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy at
The University of Texas at Arlington
December, 2019

Arlington, Texas

Supervising Committeee:

Christoph Csallner, Supervising Professor
Leonidas Fegaras
Bahram Khalili
Jeff Lei

To my beloved Tonni

ACKNOWLEDGMENTS

First and foremost, I would take this opportunity to express my sincere gratitude to my supervising Professor Dr. Christoph Csallner. This dissertation would not happen without his constant support. I am deeply grateful not only for the enlightening directions I have received but also for his sincere accommodations, which made this research collaboration meaningful and enjoyable. I know that very few are fortunate enough to have such a mentor and cannot thank him enough for the invaluable guidance I have received from him throughout my entire doctoral studies timeline and especially during my difficult times. Without his patience and encouragement none of the works included in this dissertation would have happened.

Besides my supervisor I am grateful to the rest of my PhD committee: Dr. Leonidas Fegaras, Dr. Bahram Khalili, and Dr. Jeff Lei for their encouragement and valuable feedback throughout my entire doctoral studies timeline. I am very much thankful to Dr. Taylor T. Johnson for his continual support, encouraging ideas and vital feedback which helped shaping the direction of this work.

Besides my advisers and collaborators from the academia I am very much thankful to my mentors at MathWorks: Divya Bhat, Stephen Van Kooten, Jing Shen, Dr. Akshay Rajhans, and Dr. Pieter J. Mosterman for valuable technical discussions and feedback on a significant portion of the work completed during my doctoral studies.

Lastly, I am deeply grateful to my family. I thank my beloved wife Sadia Ahmed (Tonni) for her encouragements, trust, and countless sacrifices. I am forever grateful to my parents for their love and faith.

ABSTRACT

Automated Testing of a Commercial Cyber-Physical System Development Tool Chain

Shafiul Azam Chowdhury
The University of Texas at Arlington
Supervising Professor: Christoph Csallner

Rigorous validation of commercial cyber-physical system (CPS) tool chains (e.g., MATLAB/Simulink) through automated testing is of utmost importance since tool-chain generated artifacts are often deployed in safety-critical embedded hardware. Although automated differential testing through random program generation and equivalence modulo input (EMI)-based mutation has been well studied for procedural compiler testing, applying these techniques for Simulink, the widely used commercial CPS development tool pose unique challenges, which we explore in this series of work for the very first time. To better understand real-world CPS modeling and to automatically generate Simulink models similar to those crafted by engineers and researchers, we present the largest study of Simulink models to date. Using insights from this corpus we have built the very first publicly known random Simulink model generator and differential testing framework, which has found previously unknown compiler bugs in Simulink. To further improve the automated compiler testing framework we have then explored novel EMI-based mutation techniques for Simulink models, which deal with CPS language features that are not found in procedural programs, including an explicit notion of time and zombie code which combines the properties of both dead and live code. Our resulting open source tools have discovered 21 unique Simulink bugs in various production versions to date confirmed by MathWorks Support proving bug finding capabilities of these tools. 16 of these bugs were unknown to MathWorks Support.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Model-based design of cyber-physical systems (CPS) using complex development tool chain (e.g., MathWork's Simulink) is the de-facto engineering practice in many safety-critical domains: automotive, aerospace and health care to name a few [99, 90]. Using these tool chains engineers typically design abstraction of some CPS using graphical block-diagram models to simulate their behavior and then automatically generate code and deployable executables [9, 68, 38].

Since CPS tool chain-generated artifacts are often deployed in safety-critical embedded hardware including cars and planes, quality assurance of various tool chain components (e.g., compilers, simulation solvers and code generators) through eliminating bugs is of utmost importance as tool chain defects may introduce further subtle bugs in automatically generated CPS artifacts [11].

While it would be ideal to formally verify that an entire CPS development tool chain is bug-free, unfortunately, this is practically infeasible. Moreover, it is often not possible to get a full, up-to-date, formal specification of a commercial CPS tool chain [82, 39, 8]. In contrast *differential testing*, a major contributor to software quality does not entail full formal specifications of the compiler system under test as it compares the results of two or more executions (e.g., simulations of CPS models) of the same program that are supposed to produce the same results [59].

First coined by McKeeman, randomized differential testing of compiler systems utilizes (typically automatically generated) random programs by executing and observing their outputs using two or more comparable compiler implementations or configurations [59]. If the outputs of the same program do not match then one of the compiler system implementations likely have bug(s).

As a concrete example randomized differential testing of Simulink [90], the widely used CPS development tool may involve a random Simulink model generator. The random generator automatically creates valid Simulink models which are inputs to the Simulink compiler tool chain — our compiler system under test. Next, we take one such automatically generated Simulink model to compile, execute and observe its outputs using different Simulink compiler optimization settings, e.g.. If for such a generated Simulink model we observe output divergence when executing it using different compiler optimization settings, we may confidently assert the existence of compiler bug(s) since a Simulink model's outputs should not diverge due to compiler optimization level differences.

Differential testing through random program generation has been effective in recent compiler testing projects collectively finding hundreds of bugs in commercial compiler implementations (such as GCC[1] and LLVM) that are part of CPS development tool chains [98, 75, 41, 29]. Our initial study of publicly available Simulink bug-reports also suggests differential testing as a good candidate for finding commercial CPS tool chain bugs.

While compiler testing is promising, when testing a commercial CPS tool chain like Simulink we face additional challenges beyond what is covered by testing compilers of traditional programming languages (such as Csmith creating C programs [98]), since CPS modeling languages differ significantly from traditional programming languages. Moreover, random generation of CPS models to test CPS development tool chains has to address a combination of programming paradigms (e.g., both graphical, data-flow language and textual imperative programming language in the same model) which is rare in traditional compiler testing.

Automated testing of textual programming language compiler tools is well studied, however, to the best of our knowledge no work targeted automated randomized differential testing of commercial CPS tool chains. Existing testing and verification research in the CPS domain [52, 46, 4, 34, 78, 3, 102, 10, 57] have targeted analyzing and testing the CPS models unlike this dissertation which explores finding compiler bugs by automatically testing a commercial CPS tool, namely Simulink from MathWorks [90].

Since existing approaches are not sufficient for ensuring the reliability of commercial CPS tool chains, this dissertation first proposes CYFUZZ — the first known conceptual

[1]GNU Compiler Collection

differential testing framework for testing a commercial CPS development tool and a prototype implementation to test Simulink [16] (Chapter 2). CyFuzz has a random generator, which automatically creates valid Simulink models. Its differential testing component then detects dissimilarity (if it exists) in the results obtained by executing (aka simulating) a generated Simulink model varying components of the Simulink tool chain.

CyFuzz pioneered differential testing of commercial CPS tool chains by addressing problems unique to CPS tool chain testing and presenting a prototype implementation to automatically test Simulink which rediscovered one previously known Simulink bug. However, one key CyFuzz limitation is that it cannot generate Simulink models rich in language features partly as it does not leverage any Simulink specifications. Although complete and updated formal specifications for Simulink are not available, one could still leverage the specifications made available through Simulink's official web-based documentations expressed in a semi-formal structure to generate valid Simulink models rich in language features.

Prior studies attributed generating programs rich in language feature to the success of randomized differential testing schemes [85]. Not leveraging any of the available specifications has perhaps significantly crippled CyFuzz's bug finding capabilities. Indeed, in the experiment period CyFuzz did not find any new Simulink bug.

Another key CyFuzz limitation is that it did not explore metrics to estimate how close a generated model is to those designed by CPS engineers and researchers. To the best of our knowledge no large-scale study of Simulink modeling practices is available, although large repositories of publicly available programs in major textual programming languages (e.g., Java) exist [101, 19, 37]. A large-scale corpus of publicly available Simulink models could guide a random generator to create models which tool chain developers are more likely to care about.

To address these shortcomings, next we present the first large-scale collection of public Simulink models and use the collected models' properties to guide random Simulink model generation (Chapter 3). To guide model generation we systematically collected semi-formal Simulink specifications from official documentations designing easy-to-craft parsing tools. In our experiments on several hundred models, the resulting SLforge generator was more effective and efficient than its predecessor and the most relevant competitor CyFuzz [17]. SLforge also found 11 unique bugs (9 of which new) in various Simulink production versions confirmed by MathWorks Support, which is the currently the state-of-the-art valid Simulink model generator.

In subsequent work we further extended the corpus of Simulink models, which contains about 1,000 publicly available Simulink models and is currently the largest such corpus of CPS models (Chapter 4). Besides directly enabling evaluation of random Simulink model generators, our publicly available corpus can also benefit all future Simulink model-based studies and tool development efforts.

While random program generation for compiler testing can potentially discover many unknown bugs initially, it usually requires years of engineering efforts to design well-formed program generators [23, 48]. In contrast, using relatively less engineering effort *Equivalence Modulo Input (EMI)*-based compiler validation, a recent variation of randomized differential testing has found over a thousand of bugs in major GCC and LLVM versions which were missed by random program generation (aka fuzzing) alone [48, 49, 85, 83].

EMI-based scheme mutates existing C code (aka *seed*) collected from real-world corpus or random generators to create one or more valid programs (aka *mutants*) functionally *equivalent* to the original program, modulo some input common to both the seed and its mutants. Using *differential testing* it can then automatically detect output discrepancies in the generated mutants which is a potential compiler bug indicator [59, 51].

Besides, compared to random program generators EMI-based approaches stress the optimizers and code generators well, which are generally attributed to the most vulnerable compiler components [48, 17, 82, 16]. This, along with its effectiveness in compiler bug finding motivated us to explore its applicability in CPS tool chain testing. The only related work we are aware of is SLforge, which primarily focused on random Simulink model generation and only examined a restricted mutation technique based on statically finding and then removing all components in a model, finding one EMI-bug [17].

Although EMI-based mutation via dynamically detecting and pruning dead program components and introducing modifications in live program paths have been proven effective for procedural compiler testing [48, 85, 85], to the best of our knowledge, no study has yet explored such techniques in the context of CPS tool chain testing. This is perhaps because large-scale real-world CPS model corpus [18] and random valid CPS model generators are only recently been made available, which are pre-requisites to effective EMI-based validation schemes [17, 48, 14].

EMI for CPS tool chain also introduces previously unexplored challenges in the context of EMI-based testing. For instance, CPS model components' (e.g., Simulink *blocks*) data-types can be inferred by the tool chain, which poses additional challenges compared to EMI-based mutation of C programs. Besides, EMI-based mutation of CPS programs have to deal with zombie code that combines properties of both live and dead code. Unlike prior EMI-based mutation of C and OpenCL programs, this dissertation explores EMI for zombie code for the very first time [48, 85, 49]. Also, in contrast to procedural programs, CPS models have an explicit notion of simulation and output sampling time which makes EMI-based mutation complicated. Lastly, the resultant mutants should satisfy all of the tool chain specifications so that it compiles and qualifies for effective differential testing [41].

This dissertation explores challenges in EMI-based Simulink model mutation in depth for the very first time and presents a general-purpose EMI-based automated testing framework. The resultant SLEMI tool, the very first implementation of dynamic EMI-based automated testing of Simulink models outperforms the existing and only known EMI-based scheme for Simulink, SLforge [17]. Finally, we empirically evaluate its effectiveness for testing Simulink: in our experiments SLforge found two unique bugs whereas using similar computing resource SLEMI has found 9 unique bugs confirmed by MathWorks Support in Simulink versions R2017a and R2018a. Many of the bugs found by SLEMI are out of reach of plain differential testing.

To summarize, this dissertation makes the following novel contributions for automated testing of Simulink, the widely used CPS development tool:

- To automatically test a commercial CPS development tool Simulink, we present the first known randomized differential testing framework for Simulink exploring challenges in automated generation of valid Simulink models. We analyze the feasibility of applying differential testing in finding CPS development tool bugs and present Cy-

Fuzz, the first known open source Simulink model generator and differential testing framework (Chapter 2).

- To find new compiler bugs in Simulink, we identify and address limitations of the CyFuzz tool and propose a new approach to generate valid Simulink models by construction. The resulting SLforge tool leverages official Simulink specifications and generates feature-rich Simulink models. SLforge is not only efficient compared to CyFuzz in terms of runtime but is also effective in finding new compiler bugs (Chapter 3).

- To better understand how engineers and researchers use Simulink to model CPS we conducted the largest study of publicly available Simulink models to date. Besides guiding a random Simulink model generator (e.g., SLforge) to generate realistic models, our publicly available corpus and open source tools can benefit all model-based research and tool development efforts (Chapter 4).

- Lastly, we explore a recent variation of differential testing, namely Equivalent Modulo Input (EMI)-based testing to find Simulink bugs. EMI-based mutation of Simulink models require addressing previously unexplored challenges which include handling zombie regions, an explicit notion of time and advanced modeling features (e.g., automated data-type and sample-time inference). SLEMI, our approach to EMI-based testing of Simulink is efficient compared to SLforge as it consumes relative less computing resources. Besides, many of the bugs found by SLEMI cannot be found by plain differential testing alone (Chapter 5).

All of our tools are open source, which to date have collectively found 21 unique Simulink bugs confirmed by MathWorks Support in various Simulink versions, 16 of which are previously unknown to the commercial CPS development tool vendor.

Author Contributions

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

- **Chapter 2:** CyFuzz: A Differential Testing Framework for Cyber-Physical Systems Development Environments [16]

  Proceeding 6th Workshop on Design, Modeling and Evaluation of Cyber Physical Systems (CyPhy), pages 46-60. 2016

  Authors Shafiul Azam Chowdhury, Taylor T. Johnson, and Christoph Csallner.

  Dr. Johnson and Dr. Csallner supervised the entire project shaping the research direction, reviewing the research questions and experimentation setup and significantly improving the paper. I was responsible for conducting the investigations, designing and analyzing the experiments, and implementing the CyFuzz tool for automated testing of the Simulink compiler.

- **Chapter 3:** Automatically Finding Bugs in a Commercial Cyber-Physical System Development Tool Chain With SLforge [17]

  Proceeding 40th ACM/IEEE International Conference on Software Engineering (ICSE), pages 981-992. 2018

  Authors Shafiul Azam Chowdhury, Soumik Mohian, Sidharth Mehra, Siddhant Gawsane, Taylor T. Johnson, and Christoph Csallner.

  Dr. Johnson and Dr. Csallner supervised the overall research direction, provided constant feedback on prioritizing experiments and analyses, and significantly strengthened our paper. Siddhant Gawsane was responsible for collecting and reviewing a significant portion of the corpus of Simulink models included in this work. Sidharth Mehra contributed in developing many components of the tool to collect properties of Simulink programs. Soumik Mohian was responsible for timely implementation of analyzing many Simulink model properties. I was responsible for designing and conducting the initial investigations, conducting and analyzing the experiments, and implementing core functionalities of the SLforge tool including generation of random Simulink programs.

- **Chapter 4:** A Curated Corpus of Simulink Models for Model-based Empirical Studies [18]

Proceeding 4th International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS), pages 45-48. 2018

Authors Shafiul Azam Chowdhury, Lina Sera Varghese, Soumik Mohian, Taylor T. Johnson and Christoph Csallner

Dr. Johnson and Dr. Csallner supervised the entire project from designing the Simulink model corpus to collecting and analyzing the Simulink model properties providing constant feedback and also significantly strengthened the paper. Lina Sera Varghese collected and reviewed a significant portion of the corpus. Soumik Mohian helped analyzing some model properties. I was responsible for designing the corpus, identifying the relevant metrics and developing the framework to automatically collect and analyze the metrics.

- **Chapter 5:** Equivalence Modulo Input (EMI)-based Cyber-Physical System Development Tool Chain Testing With SLEMI (paper submitted for review)

Authors Shafiul Azam Chowdhury, Sohil L. Shrestha, Taylor T. Johnson and Christoph Csallner

Dr. Johnson and Dr. Csallner supervised the entire project. Dr. Csallner helped identifying the crucial problems to address in this paper, reviewed the research questions, supervised the experimental analysis and significantly strengthened the paper. Sohil Lal Shrestha provided valuable feedback throughout the project and helped analyzing one of the research questions. I was responsible for formulating and analyzing the research questions, implementing the EMI-based mutation framework for Simulink, and conducting and analyzing the various EMI-baed mutation experiments.

Other Publications and Demonstration

- **ICSE Student Research Competition Poster:** Automatically Finding Bugs in Commercial Cyber-physical System Development Tool Chains [13]

Proceeding 40th International Conference on Software Engineering (ICSE): Companion Proceedings, pages 506-508. 2018

Author Shafiul Azam Chowdhury

Award 3rd Prize (Bronze Medal) at ACM/Microsoft Student Research Competition at ICSE 2018

- **ICSE Doctoral Symposium Poster:** Understanding and Improving Cyber-physical System Models and Development Tools [14]

Proceeding Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, pages 452-453. 2018

Author Shafiul Azam Chowdhury

CHAPTER 2

CYFUZZ: A DIFFERENTIAL TESTING FRAMEWORK FOR CYBER-PHYSICAL SYSTEMS DEVELOPMENT ENVIRONMENTS [1]

Shafiul Azam Chowdhury

Taylor T. Johnson

Christoph Csallner

Shafiul Azam Chowdhury, Taylor T. Johnson, Christoph Csallner. CyFuzz: A Differential Testing Framework for Cyber-Physical Systems Development Environments. In: Christian Berger, Mohammad Reza Mousavi, Rafael Wisniewski (eds) Cyber Physical Systems. Design, Modeling, and Evaluation. CyPhy 2016. Lecture Notes in Computer Science, vol 10107, pages 46-60. 2017. Springer, Cham.

DOI: 10.1007/978-3-319-51738-4_4

[1]Used with permission of the publisher, 2016

Abstract. Designing complex systems using graphical models in sophisticated development environments is becoming de-facto engineering practice in the cyber-physical system (CPS) domain. Development environments thrive to eliminate bugs or undefined behaviors in themselves. Formal techniques, while promising, do not yet scale to verifying entire industrial CPS tool chains. A practical alternative, automated random testing, has recently found bugs in CPS tool chain components. In this work we identify problematic components in the Simulink modeling environment, by studying publicly available bug reports. Our main contribution is CyFuzz, the first differential testing framework to find bugs in arbitrary CPS development environments. Our automated model generator does not require a formal specification of the modeling language. We present prototype implementation for testing Simulink, which found interesting issues and reproduced one bug which MathWorks fixed in subsequent product releases. We are working on implementing a full-fledged generator with sophisticated model-creation capabilities.

2.1 Introduction

Widely used cyber-physical system (CPS) development tool chains are complex software systems that typically consist of millions of lines of code [82]. For example, the popular MathWorks Simulink tool chain contains model-based design tools (in which *models* in various expressive modeling languages are used to describe the overall system under control [50]), simulators, compilers, and automated code generators. Like any complex piece of code, CPS tool chains may contain bugs and such bugs may lead to severe CPS defects.

The vast majority of resources in the CPS design and development phases are devoted to ensure that systems meet their specifications [7, 97]. In spite of having sophisticated design validation and verification approaches (model checking, automated test case generation, hardware-in-the-loop and software-in-the-loop testing etc.), we see frequent safety recalls of products and systems among industries, due to CPS bugs [95, 96, 2].

Since many CPSs operate in safety-critical environments and have strict correctness and reliability requirements [44], it would be ideal for CPS development tools to not have bugs or unintended behaviors. However, this is not generally true as demonstrated by recent *random testing* projects finding bugs in a static analysis tool (Frama-C) [24] and in popular

C compilers (GCC and LLVM) [98], which are widely used in CPS model-based design.

It would be extremely expensive or possibly even practically infeasible to formally verify entire CPS tool chains. In addition to their sheer size in terms of lines of code, a maybe more significant hurdle is the lack of a complete and up to date formal specification of the CPS tool chain semantics, which may be due to their complexity and rapid release cycles [82, 29].

Instead of formally verifying the absence of bugs in all CPS tool chain execution paths, we revert to showing the presence of bugs on individual paths (aka testing), which can still be a major contributor to software quality [59]. *Differential testing* or *fuzzing*, a form of random testing, mechanically generates random test inputs and presents them to comparable variations of a software [59]. The results are then compared and any variation from the majority (if one exists) likely indicates a bug [51]. This scheme has been effective at finding bugs in compilers and interpreters of traditional programming languages. As an example, various fuzzing schemes have collectively found over 1,000 bugs in widely used compilation tools such as GCC [98, 29, 41].

While compiler testing is promising, when testing CPS tool chains we face additional challenges beyond what is covered by testing compilers of traditional programming languages (such as Csmith creating C programs), since CPS modeling languages differ significantly from traditional programming languages. A key difference is that the complete semantics of widely used commercial modeling languages (e.g., MathWorks Simulink and Stateflow [90]) are not publicly available [82, 39, 8]. Moreover, modeling language semantics often depend on subtle details, such as two-dimensional layout information, internal model component settings, and the particular interpretation algorithm of simulators [82]. Finally, random generation of test cases for CPS development environments has to address a combination of programming paradigms (e.g., both graphical, data-flow language and textual imperative programming language in the same model), which is rare in traditional compiler testing.

Since existing testing and verification techniques are not sufficient for ensuring the reliability of CPS tool chains, we propose CyFuzz: a novel conceptual differential testing framework for testing arbitrary CPS development environments. We use the term *system* *under test (SUT)* to refer to the CPS tool chain being tested. CyFuzz has a *random model generator* which automatically generates random CPS models the SUT may simulate or compile to embedded native code. CyFuzz's *comparison framework* component then detects dissimilarity (if it exists) in the results obtained by *executing* (or, *simulating*) the generated model, by varying components of the SUT.

We also present an implementation for testing the Simulink environment, which is widely used in CPS industries for model-based design of dynamic and embedded systems [58, 79]. Although our current prototype implementation targets Simulink, the described conceptual framework is not tool specific and should thus be applicable to related CPS tool chains, such as NI's LabVIEW [63].

To the best of our knowledge, CyFuzz is the first differential testing framework for fuzzing CPS tool chains. To address the problem of missing formal semantics during model generation, we follow a simple, feedback-driven model generation approach that iteratively fixes generated models according to the SUT's error descriptions. To summarize, this paper makes the following contributions:

- To understand the types of Simulink bugs that affect users, we first analyze a subset of the publicly available Simulink bug reports (Section 2.3).
- We present CyFuzz, a conceptual framework for (1) generating random but valid models for a CPS modeling language, (2) simulating the generated models on alternative CPS tool chain configurations, and (3) comparing the simulation results (Section 2.4). We then describe interesting implementation details and challenges of our prototype implementation for Simulink (Section 2.5).
- We report on our experience of running our prototype tool on various Simulink configurations (Section 2.6), identifying comparison errors and semi-independently reproducing a confirmed bug in Simulink's Rapid Accelerator mode.

2.2 Background: Model-based CPS Design and Simulink

This section provides necessary background information on model-based development. We define the terms used for explaining a conceptual differential testing framework and subsequently relate them with Simulink.

2.2.1 CPS Model Elements

The following concepts and terms are applicable to many CPS modeling languages (including Simulink). A *model*, also known as a *block-diagram*, is a mathematical representation of some CPS [58]. Designing a diagram starts with choosing elementary elements called blocks. Each *block* represents a component of the CPS and may have *input* and *output* ports. An input port accepts data on which the block performs some operation. An output port passes data to other input ports using *connections*. An output port can be connected to more than one input port while the opposite is not true in general. A Block may have *parameters*, which are configurable values that influence the block's behavior. Somewhat similar to a programming language's standard libraries, a CPS tool chain typically provides *block libraries*, where each library consists of a set of predefined blocks.

Since hierarchical models are commonly found in industry, CyFuzz supports generating such models as well. This can be achieved by grouping some blocks of a model together and replacing them by a new block which We call a *child*, whereas the original model is called *parent*.

When simulating, the SUT numerically solves the mathematical formulas represented by the model [58]. Simulation is usually time bound and at each *step* of the simulation, a *solver* calculates the blocks' outputs. We use the term *signal* to mean output of a block's port at a particular simulation step.

The very first phase of the simulation process is *compiling* the model. This stage also looks for incorrectly generated models and raises failures for syntactical model errors, such as data type mismatches between connected output and input ports. If an error is found in the compilation phase, the SUT does not attempt simulating the model. After successful simulation, *code generators* can generate native code, which may be deployed in target hardware [82].

2.2.2 Example CPS Development Environment: Simulink

While our conceptual framework uses the above terms, they also apply directly in the context of Simulink [93]. Besides having a wide selection of built-in blocks, Simulink allows integrating native code (e.g., Matlab or C code) in a model via Simulink's S-function interface, which lets users create custom blocks for use in their models. Simulink's Subsystem and Model referencing features enable hierarchical models.

Simulink has three simulation modes. In Normal mode, Simulink does not generate code for blocks, whereas it generates native code for certain blocks in the Accelerator mode. Unlike in these two modes, the Rapid Accelerator mode further creates for the model a standalone executable. To capture simulation results we use Simulink's Signal Logging functionality as we found implementing it quite feasible. However, for cases where the approach is not applicable (see [93]), we use Simulink's sim api to record simulation data.

2.3 Study of Existing Bugs: Incorrect Code Generation

To understand the types of bugs Simulink users have found and care about, we performed a study on the publicly available bug reports from the MathWorks website[2]. We identified commonalities in bug reports, which we call *classification factors*. We limited our study to bug reports found via the search query *incorrect code generation*, as earlier studies have identified code generation as vulnerable [82, 71].

We investigated bug reports affecting Matlab/Simulink version 2015a as we were using it in our experiments. As of February 17, 2016, there were 50 such bug reports, among which 47 have been fixed in subsequent releases of the products. Table 2.1 summarizes the findings. Our complete study data are available at: http://bit.ly/simstudy

Table 2.1 shows only those classification factors that affect at least 20% of all the bug reports that we have studied. We use insights obtained from the study in our CyFuzz prototype implementation. For example, many of the bug reports (54%) are related to simulation result and generated code execution output mismatch. Thus, differential testing

[2]Available: http://www.mathworks.com/support/bugreports/

Table 2.1: Study of publicly available Simulink bug reports. The right column denotes the percentage of bug reports affected by a the given classification factor. Each bug report may be classified under multiple factors.

Classification factor	Bugs [%]
Reproducing the bug requires a code generator to generate code	60
Reproducing the bug requires specific block parameter values and/or port or function argument values and data-types	56
Reproducing the bug requires comparing simulation-result and generated code's output	54
Reproducing the bug requires connecting the blocks in a particular way	36
Reproducing the bug requires specific model configuration settings	32
Reproducing the bug requires hierarchical models	24
Reproducing the bug requires built-in Matlab functions	20

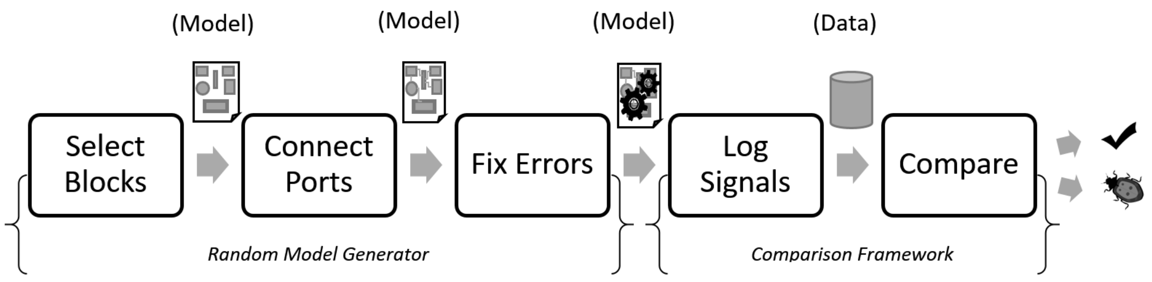

Figure 2.1: Overview of the differential testing framework. The first three *phases* correspond to the random model generator, while the rest belongs to the comparison framework.

(e.g., by comparing simulation and code execution) seems like a good fit for finding bugs in CPS tool chains. Further insight that is reflected in our tool is that it is worth exploring the large space of possible block connections (36% of bug reports) e.g., via random block and connection generation. Other insights we want to use in the future are to incorporate random block parameter values and port data-types (56%) and model configurations (32%).

2.4 Differential Testing of CPS Development Tool Chains

At a high level we can break our objective into two sub goals: creating a *random model generator* and defining a *comparison framework*. We first present a theory applicable to a conceptual CPS framework in this section. Fig. 2.1 provides a schematic overview of

CyFuzz's processing *phases*. The first three phases belong to the random model generator, and the remaining two constitute the comparison framework. The first two phases create a random model (which may violate Simulink's model construction rules). The third phase fixes many of these errors, such that the model passes the SUT's type checkers and the SUT can simulate it. If it succeeds it passes the model to the fourth phase to simulate the model in various SUT configurations and to record results. The final phase detects any dissimilarities in the collected data, which we call *comparison error* bugs.

2.4.1 Conceptual Random Model Generator

Following are details on the generator's three phases.

Listing 2.1: Select Blocks phase of the conceptual random model generator.

```
method select_blocks ($n$, $block\_libraries$):
    /* Choose $n$ blocks from the given $block\_libraries$, place the blocks
       in a new model, configure the blocks, and return the model. */
    $m$ = create_empty_model() // New, empty model
    $blocks$ = choose_blocks($n$, $block\_libraries$) // $N$ from $block\_libraries$
    for each block $b$ in $blocks$:
        place_block_in_model(m, $b$)
        configure_block($b$, $n$, $block\_libraries$)
    return $m$
```

Select Blocks.

Listing 2.1 summarizes this phase, which selects, places, and configures the model's blocks. The generator has a list of block libraries and for each library a predetermined weight. Using the weights, the $choose\_blocks$ method selects n random blocks. The value n can be fixed or randomly selected from a range. On a newly created model the generator next places each of these blocks using the $place\_block\_in\_model$ method. For creating inputs, CyFuzz selects various kinds of blocks, to, for example, provide random inputs to the model.

The $configure\_block$ method selects block parameter values and satisfies some block constraints (e.g., by choosing blocks required for placing a certain block). For creating hierarchical models, a child model is considered as a regular block in the parent model and is passed as a parameter to $configure\_block$, which calls $select\_blocks$ to create a new child model. Here n is equal to the parent model, but $block\_libraries$ may not be the same (e.g., certain blocks are not allowed in some Simulink child models).

Connect Ports.

The second phase follows a simple approach to maximize the number of ports connected. CyFuzz arbitrarily chooses an output and an input port from the model's blocks, prioritizing unconnected ports. It then connects them and continues the process until all input ports are connected. Consequently, some output ports may be left unconnected.

Listing 2.2: fix_errors tries to fix the model errors that the $simulate$ method raises; p is a SUT configuration; t denotes a timeout value.

```
method fix_errors ($m$, $p$, $attempt\_limit$, $t$):
    for $i = 1$ to $attempt\_limit$:
        $< r^p_{status}, r^p_{data}, errors > =$ simulate($m$, $p$, $t$)
        if $r^p_{status}$ is $error$:
            if fix_model($m$, $errors$) is $false$:
                return $< r^p_{status}, r^p_{data}, errors >$
        else:
            return $< r^p_{status}, r^p_{data}, errors >$
    return simulate($m$, $p$, $t$)
```

Fix Errors.

Because of their simplicity, CyFuzz's first two phases may generate invalid models that cannot be simulated successfully. The third phase tries to fix these errors. Listing 2.2 outlines the approach. It uses method $simulate$ to simulate model m up to time $t \in \mathbb{R}^+$ (in milliseconds) using SUT configuration p.

The $simulate$ output is a 3-tuple, where r^p_{status} is one of $success$, $error$, or $timed{-}out$. Note that first step of simulation is compiling the model (see Section 2.2). If m has errors, $simulate$ will abort compilation, storing error-related diagnostic information in $errors$. r^p_{data} contains simulation results (time series data of the model's blocks' outputs) if $r^p_{status} = success$.

At this point we assume that the error messages are informative enough to drive the generator. For example, Simulink satisfies this assumption. Using $errors$, fix_model tries to fix the errors by changing the model. As it changes the model this phase may introduce new errors. We try to address such secondary errors in subsequent loop iterations in Listing 2.2, up to a configurable number $attempt\_limit$. While this approach is clearly an imperfect heuristic, it has worked relatively well in our preliminary experience (as, e.g., is indicated by the low error rate in Table 2.2).

2.4.2 Conceptual Comparison Framework

Here we explore simulating a randomly generated model varying SUT-specific configuration options of a CPS tool chain, and thus testing it in two phases.

Log Signals.

If simulation was successful in the Fix Errors phase, CyFuzz simulates the model varying configurations of the SUT in this phase; let P be such a set of configurations. Using the $simulate$ method introduced in Section 2.4.1, for each $p \in P$ we calculate $< r^p_{status}, r^p_{data}, errors >= simulate(m, p, t)$ for a model m and add r^p_{data} to a set d only if $r^p_{status} = success$. We pass d to next phase of the framework. r^p_{data} should contain time series data of the output ports of the model's blocks at all available simulation steps. In the next phase, however, we use only the values recorded at the last simulation step; we leave comparing signal values at other simulation steps as future task.

Compare.

In its last phase, CyFuzz compares the recorded simulation results d obtained in the previous phase using method $compare$ (Listing 2.4). It uses method $retrieve$, which returns the signal value of a particular block's particular port at a given time instance. If the value is not available (e.g., blocks that do not have output ports do not participate in signal logging), it returns the special value *Nil*. $compare$ also uses method $latest\_time$ which returns the time of the last simulation step for a given block's particular port. If no data is available, it returns *Nil*.

Listing 2.3: Determining equivalence via tolerance limit ϵ.

```
method equiv ($p$, $q$):
  if $p$ and $q$ are $Nil$: // Missing both data points
    return $true$
  if $p$ or $q$ is $Nil$: // Missing one data point
    return $false$
  return $|p - q| < \epsilon$
```

Listing 2.4: This method compares two execution results (of model m) taken as first two arguments and throws errors if it finds a dissimilarity.

```
method compare ($r^p_{data}, r^q_{data}$, $m$):
    for each block $b$ of the model $m$:
        for each output port $y$ of the block $b$:
            $t_p$ = latest_time($r^p_{data}$, $b$, $y$)
            $t_q$ = latest_time($r^q_{data}$, $b$, $y$)
            if $equiv(t_p, t_q)$ is $false$:
                throw "Time Mismatch" error
            else if $t_p \neq Nil$:
                if $equiv(retrieve(r^p_{data}, b, y, t_p), retrieve(r^q_{data}, b, y, t_q))$ is $false$:
                    throw "Data Mismatch" error
```

Now, taking two elements from d at a time we form all possible pairs (r^p_{data}, r^q_{data}) where $p \neq q$ and apply method *compare* on them. As comparing floating-point numbers using straight equality checking is problematic [82, 35], *eqiv* (Listing 2.3) method uses a tolerance limit to determine floating-point equivalence. If *compare* reports an error, we mark

m as a *comparison error* for p, q and submit it to manual inspection.

2.5 CyFuzz Prototype Implementation for Simulink

We have developed a prototype implementation of CyFuzz mostly in Matlab. The tool continuously generates one Simulink model at a time and then passes it to the comparison framework. Source code, implementation and usage details, sample generated models, and detailed experiment results are available at: https://github.com/verivital/slsf_randgen.

Selecting and Configuring Blocks. Simulink itself has over 15 built-in libraries. MathWorks also offers toolboxes, which add to Simulink additional libraries. To date we have included in our experiments blocks from only four of these libraries, Sources, Sinks, Discrete, and Continuous. We use default parameter values for configuring most blocks. However, some Simulink blocks do not allow placing multiple instances of the same block with the same default value in a model. For these blocks we randomly choose parameter values.

Generating Hierarchical Models. Since hierarchical models are very popular among Simulink users, our prototype can generate them. Currently, the generator uses Model referencing and For each subsystems blocks to create hierarchical models. CyFuzz generates model hierarchies up to a configurable depth. In doing so it places and configures related blocks. For example, CyFuzz automatically puts input (output) related blocks in a new child model which are used to accept (return) data from (to) the parent model. The number of blocks for the top-level and child models are chosen randomly from user-provided ranges.

Fix Errors Phase. We utilize Matlab's exception handling mechanism to learn what prevented successful compilation of the model. Some information (e.g., the error type) can be directly collected from the exception. Collecting other important information, such as the actual problematic block, can be nontrivial. For example, for algebraic loop errors sometimes CyFuzz has to identify other blocks (e.g., a parent block) to fix the problem. As another example, the current CyFuzz version does not attempt to know the data types of the ports in the Connect Ports phase. Rather, it collects such information when compiling the model using diagnostic information returned by the SUT.

Models with Random Native Code. To facilitate blocks with custom behavior, Simulink allows placing native code (C, Matlab etc.) directly in models. To generate such blocks we leverage Csmith, which generates random C programs [98]. We designed simple Simulink blocks using Matlab's S-function interface that use random code generated by a customized version of Csmith. Our customized version is capable of generating many different C functions that can be called from various simulation steps. We looked for both crash errors and "wrong code errors" (similar to our comparison error). However, this is not fully integrated with CyFuzz yet.

The Comparison Framework. CyFuzz starts with varying simulation modes (see Section 2.2.2). and compiler optimization levels. For instance, "Normal mode", "Accelerator mode; optimization on", and "Rapid Accelerator; optimization off" are options to vary. Varying compilers, code generators, solver-specific settings, and other possible SUT configuration options are future work.

2.6 Experience with CyFuzz

Here we analyze our prototype implementation based on experimental results.

2.6.1 Research Questions (RQ), Experimental Setup, and Results

Throughout this work we explore the following research questions.

RQ1 Is the random model generator effective? Which portion of the generated models can the SUT compile and simulate within a given time bound?

RQ2 Using the generated models, can the comparison framework effectively find bugs (comparison errors or crashes) in the SUT ?

Table 2.2: Each row represents a separate experiment. Columns 3–6 is the percentage of blocks selected per library (e.g., experiment A chose 80% of the blocks from the Discrete library). *Error* denotes the number of models that failed to simulate. *Timed-out* denotes the models that did not complete simulation within the time bound.

Exp. Label	Total Models	Discrete [%]	Continuous [%]	Source [%]	Sink [%]	error [%]	timed-out [%]	Confirmed Bugs [%]
A	1172	80	0	10	10	9.73	0.60	0
B	1095	43	37	10	10	1.74	7.03	0
C	1449	0	80	10	10	12.01	8.63	0

Table 2.3: More information on experiments from Table 2.2. Columns 3-7 denotes the time taken by the five phases of CyFuzz. *Runtime* denotes the average time CyFuzz spent for a model.

Exp. Label	Blocks/ Model	Select Blocks [%]	Connect ports [%]	Fix Errors [%]	Log Signals [%]	Compare [%]	Runtime [sec]
A	35.00	7.85	0.64	16.00	74.55	0.96	40.37
B	34.96	6.06	0.39	16.06	76.86	0.63	51.87
C	35.05	8.09	0.51	11.02	79.58	0.80	42.51

RQ3 What is the runtime of each of CyFuzz's stages? Does the generator scale with the generated model's number of blocks?

To answer these questions we conducted experiments using Matlab 2015a on Ubuntu 14.10 and varied simulation mode (Normal vs. Accelerator) and optimizer (on vs. off) for the later mode. For the fix_errors method (Listing 2.2) we chose $attempt\_limit$ 10 and $timeout$ 12. For choosing blocks we used a traditional $O(n)$ implementation of the *fitness proportion selection* algorithm [36]. We have not included in these experiments hierarchical models or custom blocks.

Effectively Creating Random Models (RQ 1). As the experimental results in Table 2.2 suggest, our tool can generate many models that Simulink can successfully simulate. For each row in the table we have a low error and timed-out rate. This high success rate is crucial for the framework as it only uses such valid models in the tool's later comparison framework phases. We also observed that the number of errors and timed-out models varied with the selected block libraries, but we have not yet analyzed the reasons of these variations.

Effectiveness of Comparison Framework (RQ 2). We have not found new bugs yet, however, our framework reproduced an existing bug[3] and found interesting cases (see Section 2.6.2).

Runtime Analysis (RQ 3). The Select Blocks algorithm of Listing 2.1 has runtime $O(n)$, n being the number of blocks in the model and using an $O(1)$ block selection algorithm. The random model generator scales linearly with the number of blocks. But as the number of blocks grows, the number of timed-out models and errors also grow. A preliminary analysis suggests that there are relatively few distinct error causes. We group errors by their causes and fixing one cause dramatically increased the overall number of successfully executed models.

Table 2.3 indicates that the Log Signals phase uses most of the runtime. This result is not surprising, as in this phase the SUT simulates the model, generates and executes code, and logs the data, all of which are time consuming tasks.

Using Native Code/Custom Blocks.

In separate experiments we used a fixed Simulink model with a custom block created using $\mathrm{S\text{-}Function}$. We repeatedly generated random C code using a customized version of Csmith and plugged this code in the $\mathrm{S\text{-}function}$, which effectively ran the code once we simulated the model. We used different optimizer settings for GCC when compiling and were able to reproduce crash and “wrong code” bugs of GCC 4.4.3. This shows that incorporating Csmith in our framework is promising. However, more work is needed to fully utilize Csmith-generated programs and create sophisticated Simulink blocks using them. One limitation is that floating-point support in Csmith is currently still basic and can only be used for detecting crash-bugs.

[3]MathWorks has fixed this bug in a future release.

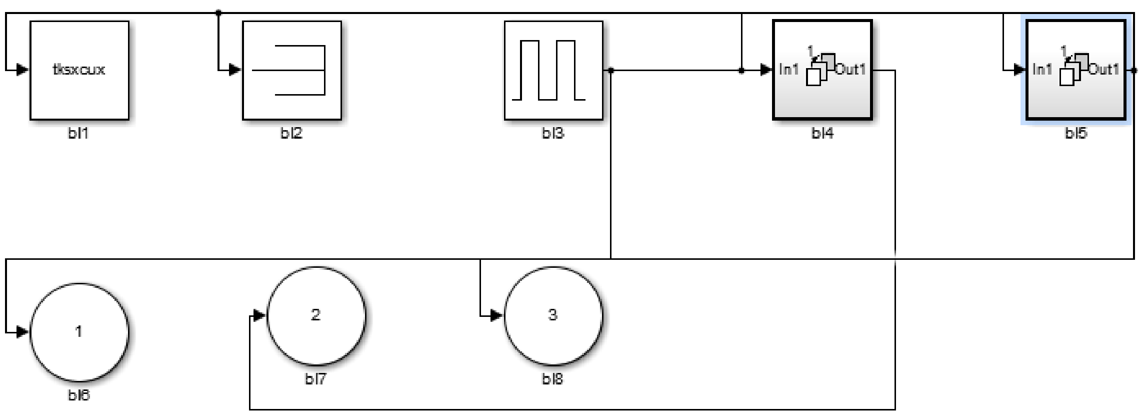

Figure 2.2: Screen-shot of generated top-level Simulink model which reproduced a bug

2.6.2 Interesting Comparison Framework Findings

Following are two interesting findings of our experiments, including one independently rediscovered confirmed Simulink bug.

Comparison Error for Models with Algebraic Loops.

In our experiments we noticed comparison errors for some models where Simulink solved algebraic loops. Investigating further we noticed that when Simulink solves an algebraic loop it is not confident of its correctness [93]. For this, we did not classify this case as a bug. CyFuzz now eliminates algebraic loops altogether rather than relying on Simulink to solve them. We note that one can use our tool to opportunistically discover such inaccuracies for models with algebraic loops and decide whether to accept Simulink's solution for solving the loops.

Bug in Simulink's Rapid Accelerator *Mode.*

In separate experiments with hierarchical models, we noticed that for a model (see Fig. 2.2) values of a Simulink Outport block are significantly different in Normal and Rapid Accelerator mode. This was detected automatically by our comparison framework. After submitting a bug report MathWorks confirmed that the case was already identified as a bug and they fixed it for later versions.

2.7 Future Work and Discussion

Our ultimate goal is to provide a full-fledged fuzz-testing framework for Simulink. Our work on CyFuzz and our prototype implementation for Simulink are thus both ongoing. Following is a sample of the opportunities for improvement.

The current prototype implementation has several limitations. Currently, the tool chooses blocks from only four built-in libraries. Incorporating additional libraries will increase the expressiveness of generated models and thus its potential for finding bugs. Also, we plan on integrating custom blocks developed using native code and perform experiments we were not able to conduct yet.

The comparison framework implementation is also not free from shortcomings. So far, we have only used various simulation modes and compiler optimization levels. However, we are interested in adding more variations (e.g. those listed in Section 2.5). Finally, CyFuzz should compare signals in multiple simulation steps, since it was also found effective in previous work [65].

2.8 Related Work

The following focuses on the most closely related work not covered by the introduction section. Existing approaches for CPS testing mostly aim at generating test cases for existing models (e.g., [34, 58]) and do not target testing of CPS tool chains. *Code generator testing* ([82, 81]) only target a relatively small component of the CPS tool chain but not an entire CPS tool chain.

Most of the compiler fuzzers perform random walks over a context-free grammar, thus mainly focusing on generating syntactically valid [41] and well typed programs in imperative languages [22, 98, 29, 42]. None of the works target data-flow languages like Simulink. We find Csmith most related to our work, which is state-of-the-art C compiler fuzzer. Csmith leverages the well-published C99 standard and can be used to test only a component of entire CPS tool chain [98]. Our test generation and comparison techniques differ fundamentally from Csmith. Conceptually, CPS tool chain fuzzing is a super-set of the schemes presented in Csmith. CPS tool chains typically contain a C compiler; thus

CyFuzz leverages Csmith as a component.

Earlier work includes a differential testing based runtime verification framework, leveraging a random hybrid automata generator [64, 65]. Other works attack code generators used in CPS tool chain. Stürmer et al. generate model taking specification of a code generator's optimization rules in graph grammar [82]. But such specifications for code generators might not be available and white-box testing in parts is undesirable [76]. Sampath et al. propose testing *model-processing tools* taking semantic meta-model of Stateflow (a Simulink component) [76]. But the approach does not scale and the complete specifications it needs are not available. In contrast, we propose the first fuzz-testing framework to test arbitrary CPS tool chains based on feasible model generation.

Many CPS model verification and safety checking approaches have been proposed [44, 61]. Recent work verifies existing SL/Stateflow (SL/SF) models by generating test inputs for these models [58, 79]. Alur et al. analyze generated symbolic traces of a SL/SF model, and combine simulation and symbolic analysis for improving coverage of given SL/SF models [46]. The *Simulink Code Inspector* compares generated code for a given model based on structural equivalence and traceability [93]. However none of these approaches describe random generation of Simulink models for fuzzing the CPS tool chain.

2.9 Conclusions

This work addresses the CPS tool chain quality problem using a differential testing scheme. Existing work either does not test CPS development tool chains or only tests small subsets. As CPS tool chains are actively developed and released, formal specification based test generation schemes are not suitable for fuzzing CPS tool chains. Rather, our approach follows a simple model generation strategy applicable to arbitrary CPS modeling languages. Starting with a random and possibly erroneous model, our generator fixes various errors in the model using diagnostic information returned by the system under test. In our experiments a high portion of the generated models could thus be executed without errors.

We also define techniques to find bugs in CPS tool chains based on simulation result comparison. The approach is effective as our prototype implementation for Simulink found interesting cases and one bug. Although our model generator is scalable and fully automatic, more work is needed to systematically search the huge space of possible data-flow models and generate those models that are likely to find bugs in modern CPS development environments.

CHAPTER 3

AUTOMATICALLY FINDING BUGS IN A COMMERCIAL CYBER-PHYSICAL SYSTEM DEVELOPMENT TOOL CHAIN WITH SLFORGE[1]

Shafiul Azam Chowdhury

Soumik Mohian

Sidharth Mehra

Siddhant Gawsane

Taylor T. Johnson

Christoph Csallner

Shafiul Azam Chowdhury, Soumik Mohian, Sidharth Mehra, Siddhant Gawsane, Taylor T. Johnson, and Christoph Csallner. Automatically Finding Bugs in a Commercial Cyber-physical System Development Tool Chain with SLforge. In Proc. 40th International Conference on Software Engineering (ICSE ‘18), pages 981-992. 2018
DOI: 10.1145/3180155.3180231

[1]Used with permission of the publisher, 2018

Abstract. Cyber-physical system (CPS) development tool chains are widely used in the design, simulation, and verification of CPS data-flow models. Commercial CPS tool chains such as MathWorks' Simulink generate artifacts such as code binaries that are widely deployed in embedded systems. Hardening such tool chains by testing is crucial since formally verifying them is currently infeasible. Existing differential testing frameworks such as CyFuzz can not generate models rich in language features, partly because these tool chains do not leverage the available informal Simulink specifications. Furthermore, no study of existing Simulink models is available, which could guide CyFuzz to generate realistic models.

To address these shortcomings, we created the first large collection of public Simulink models and used the collected models' properties to guide random model generation. To further guide model generation we systematically collected semi-formal Simulink specifications. In our experiments on several hundred models, the resulting SLforge generator was more effective and efficient than the state-of-the-art tool CyFuzz. SLforge also found 9 new bugs confirmed by MathWorks Support in Simulink versions R2015a and R2017a.

3.1 Introduction

Cyber-physical system developers rely heavily on complex development environments or tool chains, which they use to design graphical *models* (i.e., *block-diagrams*) of cyber-physical systems. Such models enable engineers to do rapid prototyping of their systems through simulation and code generation [38]. Since automatically generated native code from these data-flow models are often deployed in safety-critical environments, it is crucial to eliminate bugs from cyber-physical system tool chains [9, 68].

Ideally, one should formally verify such tool chains, since a tool chain bug may compromise the fidelity of simulation results or introduce subtle bugs in generated code [11]. However, a commercial cyber-physical system (CPS) development tool chain consists of millions of lines of code, so formal verification does not (yet) scale to such tool chains. While compilers and other CPS tool chain components remain mostly unverified, we continue to observe frequent safety recalls in various industries [95, 96, 2]. The recalls are attributed to hidden bugs in the deployed CPS artifacts themselves, in spite of spending significant efforts in their design validation and verification [7, 97].

Testing, on the other hand, is a proven approach to effectively discover defects in complex software tool chains [59]. Especially *randomized differential testing* has recently found over a thousand bugs in popular production-grade compilers (e.g., GCC and LLVM) that are part of CPS development tool chains [29, 98, 48, 75, 41]. The technique eliminates the need of a test-oracle and can hammer the *system under test* in the absence of a complete formal specification of the system under test—a phenomenon we commonly observe in commercial CPS tool chain testing [82, 39, 8]. Differential testing seems suitable for black-box testing of the entire CPS tool chain, and its most susceptible parts (e.g., code generators) in particular [82, 71]. *CyFuzz* is the first (and only) known randomized differential testing tool for CPS data-flow languages [16].

While CyFuzz initiated the work for testing CPS tool chains, more work is necessary to evaluate the scheme's capabilities, e.g., for finding bugs in Simulink that developers care about. For instance, a random model generator should generate tests with properties similar to the models people typically use, since they are more likely to get fixed by the system under test (SUT) developers. While large repositories of publicly available programs of various procedural and object-oriented programming languages exist [101, 19, 37], we are not aware of such a collection of CPS models. The existing CPS studies rely on a handful of public [69] or proprietary [78] Simulink models.

Among other shortcomings, the models CyFuzz generates are small and lack many syntactic constructs. Recent studies identified expressive test-input generation as a success-factor for compiler validation ([98, 48]). Perhaps due to its inability to generate large tests with rich language features, CyFuzz has not found previously unknown bugs. Furthermore, CyFuzz essentially generates *invalid* models and iteratively fixes them until the SUT can compile and simulate them without error or exception. However, this heuristic approach required several time-consuming iterations and did not use Simulink specifications, which are available publicly in natural language.

To address these shortcomings, we have conducted the first study of a large number of public Simulink models. The size of many of these models is larger than the average size of models used in industry. From the collected models we obtain properties that are useful for targeting a random Simulink model generator. Our model collection is publicly available and may eliminate the nontrivial overhead of artifact-collection in future studies.

Next, extending CyFuzz, we present SLFORGE, a tool for automatically generating models with advanced Simulink language features. The goal is that the SLforge-generated models are similar to the collected public models. Improving on CyFuzz's undirected random model generation approach, SLforge can generate models more efficiently, by consulting available (informal) Simulink specifications.

Finally, we provide the first approach to *Equivalent modulo input* (EMI) testing in CPS development tool testing [48]. SLforge creates EMI variants from the random models it generates and uses them in the differential testing setup. During an approximately five months long testing time, we found and reported 13 bugs overall, MathWorks Support confirmed 11 of them, of which 9 were previously unknown. To summarize, the paper makes the following major contributions.

- To better target a random CPS model generator, we conduct the first large-scale study of publicly available Simulink models. A significant portion of these models are of size and complexity that are comparable to models used in industry.

- We identify problems in the existing CPS random model generator CyFuzz and design solutions that directly led to the discovery of new bugs in the Simulink tool chain.

- Finally, by comparing it with CyFuzz, we evaluate SLforge's efficiency and bug-finding capability.

3.2 Background

This section provides necessary background information on CPS dataflow models, the major commercial CPS tool-chain Simulink, the state-of-the-art differential CPS tool-chain testing tool CyFuzz, and EMI-based differential testing.

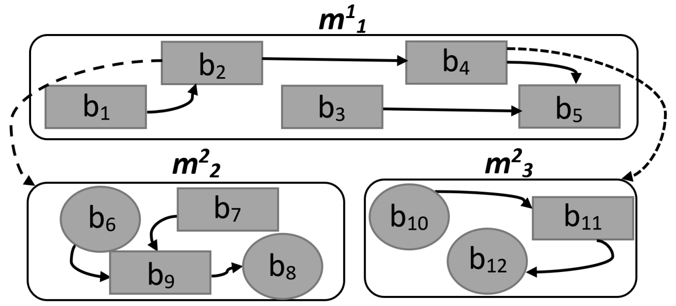

Figure 3.1: Example hierarchical CPS model: Rounded rectangle = model; shaded = block; oval = I/O; solid arrow = dataflow; dashed arrow = hierarchy.

3.2.1 CPS Data-flow Models And Simulink

While in-depth descriptions are available elsewhere [93], the following are the key concepts. In a CPS development tool (e.g., Simulink), a user designs a CPS as a set of dataflow *models*. A model contains *blocks*. A block accepts data through its *input ports*, typically performs on the data some operation, and may pass output through its *output ports* to other blocks, along *connection* lines. Simulink specifies which port (of a block) supports which *data-types*.

In a connection, we name the block sending output *source* and the block receiving data a *target*. Output ports are numbered starting with 1. Input port numbering starts with 0, where 0 denotes a special port (e.g., the Action port of the $\mathrm{If\ Action}$ block). In addition to such explicit connections, using From and Goto blocks, one can define implicit (*hidden*) connections [73, 68].

Commercial CPS tool chains offer many *libraries* of built-in blocks. Besides creating a CPS from built-in blocks, one can add *custom blocks* and define their functionality via custom “native” code (e.g., in Matlab or C, using Simulink’s *S-function* feature). Most blocks have user-configurable *parameters*.

More formally, block $b \in B$ and connection $c \in C$ may be part of model $m \in M$. Then a flat (non-hierarchical) model is a tuple $\langle B, C\rangle$ where $m.B$ and $m.C$ denote the model’s blocks and connections. Each connection is a tuple $\langle b_s, p_s, b_t, p_t\rangle$ of source block b_s, source output port p_s, target block b_t, and target input port p_t. While a Simulink connection may have multiple targets, we break such a multi-target connection into multiple (single-target) connection tuples, without losing expressiveness.

For hierarchical models we list a model m^i at hierarchy level i with its n direct child models as $m^i[m^{i+1}_k, \ldots, m^{i+1}_{k+n-1}]$. The Figure 3.1 example $m^1_1[m^2_2, m^2_3]$ has m^1_1 as its *top-level* model. m^2_2 and m^2_3 are m^1_1's *child* models at hierarchy level 2. The dashed arrow starting at b_2 indicates that in the m^1_1 model b_2 is a *placeholder* for the m^2_2 model. Block b_1 sends data to m^2_2, where b_6 receives it. Block b_8 sends data back to b_4 in m^1_1.

Example placeholders are the $\mathrm{subsystem}$ and $\mathrm{model\ reference}$ blocks. A child model m's semantics are influenced by m's *hierarchy-type* property T_h, which depends on m's configuration, the presence of specific blocks in m, and on m's placeholder block-type.

After designing a model in Simulink, users typically *compile* and *simulate* it. In simulation, Simulink numerically solves the model's mathematical relationships established by the blocks and their connections and calculates various non-dead (Section 3.4.3) block's outputs according to user-requested sample-times and inferred context-dependent *time steps*, using built-in *solvers* [58]. Simulink offers different *simulation modes*. While in *Normal* mode Simulink "only" simulates blocks, it also emits some code for blocks in *Accelerator* mode, and a standalone executable in *Rapid Accelerator* mode.

An input port p of block b is *Direct Feed-through* if b's output depends on values received from p. The parent to child model relation is acyclic. But within a model Simulink permits *feedback loops* (circular data flow). During compilation Simulink may reject a model if it fails to numerically solve feedback loops (aka *algebraic loops*). Simulink also supports explicit control-flow, e.g., via If blocks. An If ("driver") block connects to two $\mathrm{If\ Action}$ subsystem blocks, one for each branch, via the $\mathrm{If\ Action}$ block's Action port.

3.2.2 Testing Simulink With CyFuzz

CyFuzz is the first known differential testing framework for CPS tool chains [16]. The framework has five phases. The first three phases create a random model and the last two phases use the model to automatically test a SUT.

Specifically, starting from an empty model, (1) the *Select Blocks* phase chooses random blocks and places them in the model. (2) The *Connect Ports* phase connects the blocks' ports arbitrarily, yielding a model the SUT may reject, e.g., due to a type error. For example, an output port's data type may be incompatible with the data type of the input port it connects to. (3) CyFuzz iteratively fixes such bugs in the *Fix Errors* phase, by responding to the SUT's error messages with corresponding repair actions. This "feedback-driven model generation" approach, despite being an imperfect heuristic, can fix many such model errors.

Once the SUT can compile a randomly generated model, (4) CyFuzz's *Log Signals* phase simulates the model under varying SUT options. The key idea of differential testing is that each such simulation is expected to produce the same results. This phase records the output data (aka *signals*) of each block at various time-steps. CyFuzz uses different Simulink simulation modes, partly to exercise various code generators in the tool chain. Finally, in addition to SUT crashes, (5) the *Compare* phase looks for signals that differ between two simulation setups, which also indicate a bug.

CyFuzz categorizes its generated models into three groups: (1) *success*: models without any compile-time and runtime errors, (2) *error*: models with such errors, and (3) *timed-out*: models whose simulation did not complete within a configured time-out value. Although CyFuzz pioneered the differential testing of Simulink using randomly generated models, it did not find new bugs, perhaps since the generated models are small and simple (using only four built-in libraries and lacking advanced modeling features). Also, CyFuzz does not use Simulink specifications and solely relies on iterative model correction.

3.2.3 EMI-based Compiler Testing

Equivalent modulo input (EMI) testing is a recent advancement in differential testing of compilers for procedural languages [48]. Complementing plain differential testing, EMI found over one hundred bugs in GCC and LLVM [11]. The idea is to systematically mutate a source program as long as its semantics remain equivalent under the given input data. Engineering basic mutators is relatively easy and the overall scheme can effectively find bugs, when combined with a powerful random generator that can create expressive test inputs (e.g., Csmith [98]).

In its original implementation, EMI mainly leverages Csmith, which generates random C programs that do not take user inputs. A given compiler in a given configuration can then be expected to produce programs that yield the same result on all EMI-mutants of a given source program. The initial implementation proved very effective and found 147 bugs in production-grade C compilers such as GCC and LLVM.

3.3 Public Simulink Model Collection

To understand the properties of CPS data-flow models designed by both researchers and engineers, we conducted the first large study of Simulink models. The largest earlier Simulink model collection we are aware of contains some 100k blocks [26]. However, these models are company-internal and thus not available for third-party studies. In contrast, our collection consists of some 145k blocks, which are all publicly available (some require a standard Simulink license as they are shipped with Simulink).

For context, earlier work reports that at Delphi, a large industrial Simulink user, an average Simulink model consists of several hundred blocks [52]. Of the models we collected, 35 consist of more than 1,000 blocks, which is larger than an average model at Delphi.

3.3.1 Model Collection and Classification

For this study, we used the Simulink configuration our organization has licensed, which includes the base Simulink tool chain and a large set of libraries. This configuration includes the latest Simulink version; the project web page lists the detailed configuration [5]. However, with this configuration, we could directly compile only just over half of the collected models, as the remaining ones required additional libraries that were not part of our configuration [5].

Tutorial

This group consists of official Simulink tutorial models from MathWorks[2]. We manually reviewed the models and their descriptions in the Automotive, Aerospace, Industrial Automation, General Applications, and Modeling Features categories and excluded what we considered toy examples. We also included here the Simulink-provided domain-specific

[2]Available: https://www.mathworks.com/help/simulink/examples.html

library we had access to, i.e., Aerospace. An example model from this group is *NASA HL-20* (1,665 blocks), which models "the airframe of a NASA HL-20 lifting body, a low-cost complement to the Space Shuttle orbiter" [89].

Simple and Advanced

We collected models from both major open source project hosting services for Simulink, GitHub and Matlab Central. (1) We used the GitHub search page for keyword search ("Simulink") and file extension search (Simulink extensions .mdl and .slx). (2) On Matlab Central[3] we filtered results by "content type: model" and considered only those repositories with the highest average ratings (27 projects) or "most downloads" count in the last 30 days (27 projects).

To distinguish toy examples from more realistic models, we labeled the GitHub projects no user has forked or marked a favorite as Simple and the rest as Advanced. For the Matlab Central projects, we manually explored their descriptions and labeled those that demonstrate some blocks' features or are academic assignments as Simple and the rest as Advanced.

As an example, a model from the *Grid-Connected PV Array* project is "a detailed model of a 100-kW array connected to a 25-kV grid via a DC-DC boost converter" created by a senior engineer at Hydro-Quebec Research Institute (IREQ) [70]. It has 1,320 blocks. We classified it as Advanced.

Other

This group consists of models we obtained from academic papers (5 models), the academic research of colleagues (7 models), and Google searches (16 models). An example is the *Benchmarks for Model Transformations and Conformance Checking* released by engineers at Toyota Technical Center California [43]. It has 208 blocks.

[3]Available: https://www.mathworks.com/matlabcentral/fileexchange

Table 3.1: Overview of collected public models: Total number of models (M); models we could readily compile without extra effort (C); hierarchical models (H); total number of blocks and connections.

	Group	M	C	H	Blocks	Connect.
t	Tutorial	41	40	40	10,926	11,541
s	Simple	156	99	136	7,187	7,121
a	Advanced	167	66	165	118,632	116,608
o	Other	28	14	21	8,317	9,577
	Total	391	219	362	145,062	144,847

3.3.2 Model Metrics

In this study, we focus on those model properties that are relevant for constraining a random model generator to models that are representative of realistic CPS models. Our Matlab-based tool we used to collect the following metrics is freely available on the project site [5]. The collected metric values are shown as box-plots with min-max whiskers.

Number of Blocks and Connections

Blocks and connections are the main elements of Simulink models and are counted widely [67, 52]. We have included the contents of masked blocks [93] in the parent model's count. Next, we count the total number of blocks and connections at a particular hierarchy level up to hierarchy level 7.

Our connection-count metric does not include hidden connections. For connections with multiple target ports, we count the connections' target ports. Perhaps not surprisingly, Simple models are smaller (and Advanced models are larger) than models of the other groups (Figures 3.2a and 3.2b), since we manually reviewed and classified them in this class.

Hierarchy Depth

Since industrial models are frequently organized as a hierarchy, we measured how deep these hierarchies are. We treated both subsystems and model reference blocks as adding a hierarchy level. Most of the collected models are indeed hierarchical (i.e., 362/391 models).

But the median maximum hierarchy depth did not extend five across all model groups.

More surprising were the distribution of blocks and connections across hierarchy levels (Figures 3.3a and 3.3b). These numbers were rather similar across hierarchy levels. Overall, the number of blocks and connections in each hierarchy level were small, as denoted by the small median value.

Library Participation

This metric identifies the library each model block comes from. For example, do models mostly consist of built-in blocks or do they instead contain mostly custom blocks? If we cannot resolve a block's library (i.e., due to Matlab API limitations), we record the block's library as *other*.

Fig. 3.4 suggests that only a small portion of the blocks are custom ("User_Defin"). Across all four groups, Ports & Subsystems and Math Operations were the two libraries used most frequently. SLforge thus supports these libraries (among others, see Section 3.4.1), and automatic custom block generation. We also noted a high contribution from the Signal Routing library using From and Goto blocks, which enables establishing hidden data-flow relationship (Section 3.2.1).

Requested Simulation Duration

This metric captures the total simulation time *requested* by a given model (not the actual CPU time spent in simulating it). Most of the models (except those from the Other group) used the default simulation duration value of 10 seconds (Fig. 3.2d). Consequently, we ran simulations using this default value in our experiments, and have not experimented with other possible values yet.

3.4 SLforge

To address the shortcomings in the state-of-the-art differential testing framework for CPS tool chains, this section describes the design of SLforge. Fig. 5.2 gives an overview of the SLforge's seven main phases.

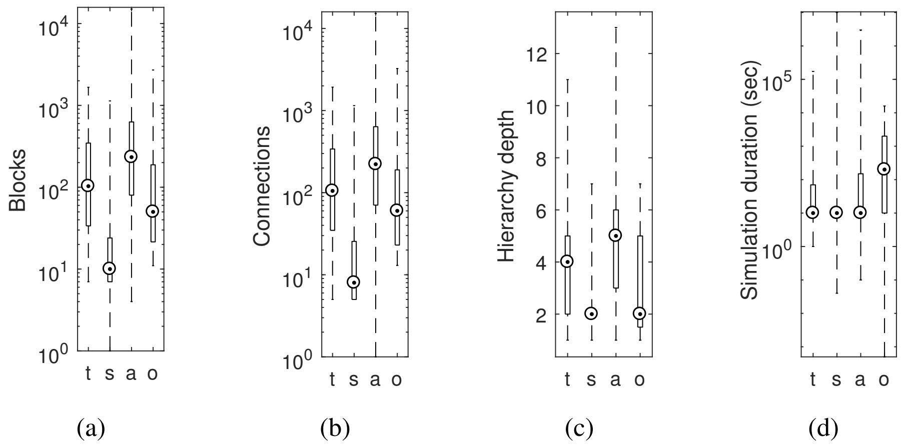

Figure 3.2: Collected public models: Total blocks (a), connections (b), maximum hierarchy depth (c), and requested simulation duration (d).

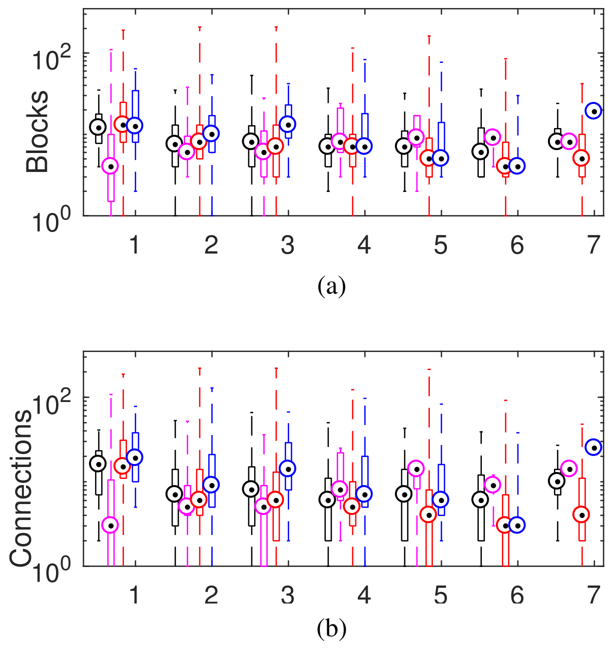

Figure 3.3: Collected public models: Blocks (a) and connections (b) by hierarchy-level, grouped by model group (t, s, a, and o).

3.4.1 Gathering Semi-Formal Specifications

CyFuzz heavily relies on its Fix Errors phase to repeatedly compile and simulate a model and iteratively repair errors based on the content of Simulink-provided error messages. Instead of this time-consuming iterative process, SLforge aims at generating a valid model

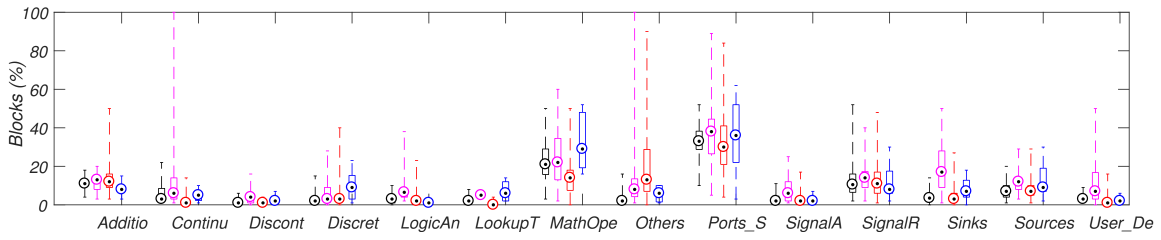

Figure 3.4: Collected public models: Distribution of blocks across libraries (shortened to the first 7 letters), each from left to right: Tutorial, Simple, Advanced, and Other.

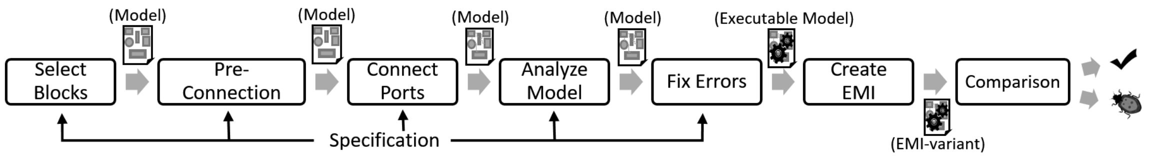

Figure 3.5: Overview of SLforge's main phases. For space, we merged CyFuzz's last two phases into the single *Comparison* phase.

in the first place, if given the language specifications. Of course, the challenge (which motivated CyFuzz's iterative process) is that there exists no complete, up to date, formal Simulink specification.

Design Choice

Simulink specifications are available as informal and semi-formal descriptions of Simulink behavior, mainly from the various Simulink web sites. From our experiments with CyFuzz, we hypothesized that many of the iterations in the Fix Errors phase are due to block data-type inconsistency and fixing algebraic loops. Besides, in our CyFuzz experiment (Section 3.5.1) the most frequent error was block *sample time* inconsistency. We collected specifications to both address these issues and to enable creating large hierarchical models (as Simulink users prefer this modeling choice).

So far, we have collected data-type support and block-parameter specifications for all built-in libraries. Other language specifications (Section 3.4.2) are often block and library specific. Since collecting the entire Simulink language specification would be overwhelming, we collected specifications for blocks from the most-used libraries. Concretely, SLforge supports blocks from Math Operations, Ports and Subsystems, Discrete, Continuous, Logic and Bit Operations, Sinks and Sources libraries. This list also covers the

CyFuzz-supported libraries and thus helps ease evaluating SLforge.

Collection Process

Using little engineering effort, SLforge's regular expression based parser parsed block data-type and parameter specifications for all built-in blocks. However, due to the limitation of the parser and Simulink's free-form specification style, SLforge can only collect parts of some specification. E.g., for three different ports of the $\mathrm{Variable\ Time\ Delay}$, $\mathrm{Discrete\ Filter}$ and Delay blocks, the Direct Feed-through property (Section 3.2.1) is described as "Yes, of the time delay (second) input", "Only when the leading numerator coefficient does not equal zero" and "Yes, when you clear Prevent direct feedthrough" respectively [93].

3.4.2 Use of Semi-Formal Specifications

Since complete and updated formal specifications for Simulink are not publicly available, existing work relies on a subset of Simulink operational specifications, which are manually crafted and possibly outdated [39, 8]. Unlike these approaches, we explored collecting specifications directly from official Simulink documentations automatically, using easy-to-engineer parsers. Such parsers can automatically update the collected specifications when new versions of the SUT are released, given the structure of the specifications go under minor or no change. From our experience with recent versions of Simulink specifications (R2015a-R2017a), the specifications SLforge collects indeed had minor structural changes.

Although SLforge parses specifications automatically and stores them using internal data-structure, for the aid of discussion, we introduce a few notions in this section. Extending the notation of Section 3.2.1, function T_b returns a block's Simulink *block-type*. For example, for each Simulink If block, T_b returns If. Next, the $valid$ predicate indicates if the Simulink type checker and runtime system accept a given model as legitimate, i.e., when there are no compile or run-time exceptions. Now we can express (part of) the Simulink specification as a formula or specification *rule*. Given such a rule $\delta \in \Delta$, we denote with $m \vDash \delta$ that model m *satisfies* (i.e., complies with) the rule. We observe that a valid model satisfies all (collected) specification rules ($\forall \delta \in \Delta : valid(m) \rightarrow m \vDash \delta$).

Select Blocks Phase

To support built-in libraries, SLforge uses specifications in this phase. Specifically, SLforge-generated models satisfy the following rules by construction. For example, using usual set cardinality notation, Equation 3.1 ensures that for each If block, the model has two $\mathrm{IfAction}$ subsystem blocks (one each for the *if* and *else* branch).

$$\begin{aligned} & 2 * |\{\, b_1 \in m.B : \, T_b\,(b_1) = If \,\}| \\ & = |\{\, b_2 \in m.B : \, T_b\,(b_2) = If Action \,\}| \end{aligned} \tag{3.1}$$

By parsing Simulink documentation, SLforge obtains a set S of blocks from the Sinks and Source libraries that are only valid in the top-level model, as enforced by Equation 3.2. Similarly, Equation 3.3 restricts using illegitimate blocks in non-top-level models, depending on the hierarchy-type property of the model (Section 3.2.1), using predicate $supports$. The predicate holds only when model m^i's hierarchy-type (first argument) allows block b in m^i, based on b's block-type (second argument).

$$\forall b \in m^i.B\text{: } (i > 1) \rightarrow (\, T_b\,(b) \notin S) \tag{3.2}$$

$$\forall b \in m^i.B\text{: } (i > 1) \rightarrow \; supports(\, T_h\,(m^i),\, T_b\,(b)) \tag{3.3}$$

$$\begin{aligned} \forall b \in m.B\text{: } & (\,(\, T_h\,(m) \in W) \wedge \; stime(b) \; = \; stime(\, driver(m)\,)\,) \\ & \vee \; (\, T_h\,(m) \notin W \wedge \; st(\, T_b\,(b), b)\,) \end{aligned} \tag{3.4}$$

Equation 3.4 configures each block's sample time property $stime$. When used in hierarchical model m of a W-listed hierarchy-type, block b's sample time should match the sample time of the model's driver (Section 3.2.1). In all other cases, we use predicate st, which holds only when the sample time property of block b is properly configured according to its block-type. To enforce such rules, SLforge propagates information from parent to child models.

Pre-connection and Connect Ports Phases

While CyFuzz uses a random search to connect unconnected ports and relies on later phases to recover from illegal connections, SLforge adds connections correctly by construction, by satisfying the following rules.

$$\forall c \in m.C\colon (\, T_b\,(c.b_t) = \mathit{IfAction}) \land (c.p_t = 0) \rightarrow (\, T_b\,(c.b_s) = \mathit{If}) \tag{3.5}$$

$$\begin{aligned}\forall c \in m.C\colon (\; T_b\,(c.b_s) = \mathit{If}) \rightarrow (\, T_b\,(c.b_t) = \mathit{IfAction}) \land (c.p_t = 0)\\ \land \,|\,\{\, c_2 \in m.C : c_2.b_s = c.b_s \land c_2.p_s = c.p_s \,\}\,| = 1\end{aligned} \tag{3.6}$$

Equation 3.5 and Equation 3.6 together specify the control-flow from an If block to its If Action blocks. Specifically, each If block output port is connected to a single (Equation 3.6) If Action block Action port.

Analyze Model Phase

On the current model state, SLforge now removes algebraic loops and assigns data-types. Instead of querying a *disjoint-set* data structure every time SLforge connects two blocks to detect whether connecting them will create a cycle, we detect them later in this phase using a single graph traversal *remove_algebraic_loops* (Listing 3.1) on each of the child models and on the top-level model. In contrast, CyFuzz relies on Simulink built-in functions to fix algebraic loops; SLforge discovered a previously unknown Simulink bug in these features (Section 3.5.3). Specifically, SLforge identifies *back-edges* and interrupts them with Delay blocks [21]. Since this process changes m, SLforge ensures that the model remains valid. For example, to ensure that the model satisfies the rules in Equation 3.5 and Equation 3.6, instead of placing a Delay block between an If and an If Action block, Listing 3.1 places it before the If block.

Listing 3.1: Removing possible algebraic loops from a model. color(b) denotes a block's visit-status via do_dfs method: white=unvisited; gray and black: visited.

```
method remove_algebraic_loops ($m$):
  $F$ = new set /* stores problematic blocks */
  for each block $b \in m.B$: set WHITE as $color(b)$
  for each block $b \in m.B$:
    if $color(b)$ = WHITE: do_dfs($m$, $b$, $F$)
  for each block $b$ in $F$:
    $s$ := get affected source block for $b$
    get and remove affected connection between $s$ and $b$
    $d'$ := add new $Delay$ block in $m$
    connect from $s$ to $d'$ and from $d'$ to $b$


method do_dfs($m$, $b$, $F$):
  set GRAY as $color(b)$
  for each connection $c \in m.C$ where $c.b_s = b$:
    if $color(c.b_t)$ = WHITE: do_dfs($m$, $c.b_t$, $F$)
    else if $color(c.b_t)$ = GRAY:
      if $c.p_t = 0$: add $b$ in $F$ else: add $c.b_t$ in $F$
  set BLACK as $color(b)$
```

After removing possible algebraic loops, SLforge propagates data-type information, to eliminate data-type mismatches. While CyFuzz compiled a model with Simulink repeatedly in the Fix Errors phase to identify data-type inconsistencies between connected blocks, SLforge fixes such errors in linear time using a single graph traversal.

Specifically, SLforge places every block whose output data-type is initially known (Non-Direct Feed-through and blocks from Source library) in a set and starting from them, runs a depth-first search on the graph-representation of the model. Data-type information is then propagated to other blocks along the connections from blocks with known output data types, using forward propagation. E.g., consider connection $c \in m.C$ and say we are currently visiting block $c.b_s$ in the depth-first search. If the data-type at $c.p_s$ is not supported by $c.p_t$ as per specification, we add a $\mathrm{Data\text{-}type\ Conversion}$ block between the ports.

3.4.3 EMI-testing

As recent work suggests that EMI-testing is promising for compiler testing [48], we explored this direction for Simulink. EMI-testing for Simulink could take many forms. For example, one could extend a model with blocks and connections that remain dead under the existing inputs. As another example, one could statically remove some of the dead blocks.

In this work we infer an EMI-variant from a given randomly generated model, by removing all blocks that are dead. We approximate the set of dead blocks statically, using Simulink's *block reduction* feature [93]. This approach differs from the original EMI implementation ([48]) in the sense that we collect the dead-block information statically, while [48] dynamically collected code coverage information. We chose the static approach as it required minimal engineering effort.

In our experiments, we noted that CyFuzz connects all output ports to certain Sink blocks. The goal was to guarantee all blocks' participation during simulation, which allowed to use Simulink's *Signal Logging* feature to record every block's outputs. Consequently, CyFuzz-generated models do not have many statically dead blocks. To let EMI-testing remove larger parts of the generated model, SLforge leaves random output ports unconnected.

3.4.4 Classification of Bugs

SLforge automatically detects Simulink crash or unresponsiveness (which we categorize as *Hang/Crash Error*) and only reports if it is reproducible using the same model. Besides crash, we discuss the types of bugs in following two directions:

Compile Time vs. Runtime

SLforge discovers some bugs during (or before) compiling the model; we categorize these as *Compile Time* bugs. In the event of compilation error, SLforge reports a bug if the error is not expected. For example, SLforge expects no data-type inconsistency error when generating type-safe models. SLforge detects some specification-mismatch bugs even before compiling, since we call various Simulink APIs to construct a model before compiling it. During this process, SLforge reports a bug when Simulink prevents it from setting a valid block parameter (according to the specification). Lastly, SLforge detects bugs when simulating the model — which we categorize as *Runtime* bugs.

Essential Feature

Here we discuss bugs based on the *essential* generator/differential testing feature that helped discovering them. We attribute *Hierarchy* to a bug if SLforge can reproduce the bug only by creating hierarchical models. Next, intuitively, SLforge attributes *Specification* to a bug when it identifies Simulink specification mismatches. Finally, like CyFuzz, SLforge identifies *Comparison* bugs by simulating a model varying SUT options (Section 3.2.2). As a special case, SLforge attributes *EMI* to a bug if some EMI-variant of a successfully simulated model does not compile, or results in comparison error when after-simulation signal data of the EMI-variant is compared with the original model or with other EMI-variants.

3.5 Evaluation

In this section we pose and explore the following relevant research questions.

RQ1 Can SLforge generate models systematically and efficiently in contrast to CyFuzz?

RQ2 Can SLforge generate feature-rich, hierarchical models in contrast to CyFuzz?

RQ3 Can SLforge effectively test Simulink to find bugs in the popular development tool chain?

To answer these research questions, we implemented SLforge on top of the open-source CyFuzz implementation. In the evaluation, we ran SLforge on 64-bit Ubuntu 16.04 virtual machines (VM), of 4 GB RAM and 4 processor cores each. We have used two identical host machines (Intel i74790 CPU (8 cores) at 3.60 GHz; 32 GB RAM each). When measuring runtime and other performance metrics (RQ1), we ran SLforge on each of the otherwise

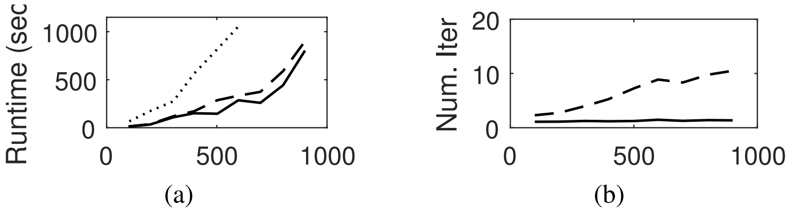

Figure 3.6: Runtime on valid models by model size given in blocks: (a) Average runtime of model generation; (b) Average number of required iterative fixes. Solid = SLforge; dashed = SLforge without specification usage and analyses; dotted = CyFuzz.

idle host machines (one VM per host). To find bugs (RQ3) we ran up to five VMs on each of these two hosts.

3.5.1 SLforge Generates Models More Systematically and Efficiently (RQ1)

To compare SLforge's new phases with CyFuzz in terms of efficiency and bug-finding capabilities, we conducted three experiments, each generating 160 models. To compare with CyFuzz, the experiments used blocks from four of the CyFuzz-supported libraries (i.e. Sources, Sinks, Discrete, and Constant). In the first two experiments we used SLforge: (1) enabling specification usage and analyses in *Exp.S+* and (2) disabling them in *Exp.S-*, and (3) in the third experiment *Exp.CF* we used CyFuzz. Across these three experiments we kept the other generator configuration parameters constant. As time-out we chose 1,200 seconds.

We compared the average time taken to generate a valid model (i.e., from Select Blocks to Fix Errors, inclusively). We also measured the number of iterative model-correction steps (*Num. Iter.*) in the Fix Errors phase. However this metric was not available in Exp.CF. In each of these experiments we started with generating 100 blocks (on average) per model, and gradually increased the average number of blocks (by 100 in each step, using 20 models in each step), up to 800 blocks/model on average. To approximate the bug-finding capability of these three setups, we counted the total number of unique bugs found in each of these experiments. Data in Both SLforge versions had a lower average runtime than CyFuzz (Fig. 3.6a).

As the number of blocks increases, Exp.S- needs more time than Exp.S+ to generate success models. When we configured CyFuzz to generate models having 700 (or more) blocks on average, it failed to generate any valid models.

Similarly, SLforge needs fewer iterations in the Fix Errors phase (Fig. 3.6b) in Exp.S+. Moreover, this value remains almost constant in Exp.S+. However, perhaps not surprisingly, the number increases with the number of blocks in Exp.S-. This result indicates that SLforge generates models more systematically as it reduces the dynamic error-correction steps significantly.

Next, we examine how the changes in SLforge affect the tool's bug-finding capability. The total number of unique bugs found in Exp.S+, Exp.S- and Exp.CF are 4, 1, and 0, respectively. Exp.S+ found the same bug discovered in Exp.S-, and found 3 more bugs. While investigating, we observed that having specifications enabled SLforge finding those bugs, since without the specifications, SLforge could not determine whether it should report a bug given an error message returned by Simulink. As an example, consider compiling a model with Simulink which results in a data-type inconsistency error between two blocks in the model. Leveraging the data-type support specifications of the two blocks, SLforge can report a bug when it does not expect any data-type inconsistency between the blocks.

Finally, a crucial step to efficiently generate large hierarchical models is to eliminate the algebraic loops from them. Simulink also rejects simulating some models in Accelerator mode in the presence of algebraic loops, which prevents differential testing using those model. As discussed before, CyFuzz depends on Simulink's buggy APIs to remove such loops, whereas SLforge eliminates the loops in the Analyze Model phase.

3.5.2 SLforge Generates Large, Feature-rich Models (RQ2)

In this experiment we compare various properties of the SLforge-generated models to the models used in our study of public models (excluding the Simple models), and to CyFuzz-generated models. First, to compare SLforge with CyFuzz, we configured SLforge and CyFuzz to generate models with hierarchy depth 7, as this value is slightly larger than the median values for all model classes. For time-out parameter we chose 800 seconds and generated 100 models using each of the tools. CyFuzz's success (of generating valid models) rate dropped to 2%. We hypothesize that CyFuzz is not capable of generating such large hierarchical models and reduced the maximum hierarchy depth to 3. After generating 100 models in this configuration, CyFuzz achieved a 12% success rate. In contrast, SLforge achieved a 90% success-rate with 7 depth.

In this experiment, SLforge generated models with an average of 2,152 blocks (median: 1,776), an average of 2,544 connections (median: 2,107), and an average hierarchy depth of 7 (median: 7). Both the average and median values of these properties are larger than (but still within the same order of magnitude as) the values we observed in the collected public models. SLforge-generated models are in this sense similar to the collected public models.

3.5.3 SLforge Found New Bugs in Simulink (RQ3)

To answer RQ3, SLforge continuously generated models and tested Simulink for approximately five months. Throughout the experiments, configuration options for SLforge varied and became applicable once we implemented a particular feature. In all of these experiments we used the Normal and Accelerator simulation modes in the comparison framework.

We have reported almost all of the SLforge-suspected bugs to MathWorks except two cases where we had associated the bug to an implementation error. For each of the reported cases, MathWorks has indicated if it considers the case a bug. For this work we mark a report as a *false positive* if MathWorks considers the case a non-bug. Our rate of false-positive is low: 2/13 reports.

Table 5.1 summarizes all the bugs we have reported. MathWorks Support has confirmed 11 of our reported issues as unique bugs, of which 9 are new bugs[4]. Following are details of a representative subset of the confirmed bugs, including SLforge-generated models we manually reduced to fit the space and aid bug-reporting. Automated test-case reduction is part of future work. These models are freely available [5].

[4]MathWorks has fixed one of the bugs (TSC 02515280) in a future release.

Table 3.2: SLforge-discovered issues and confirmed Simulink bugs: *TSC* = *Technical Support Case* number from MathWorks; *St* = status of bug report (*NB* = new bug, *KB* = known bug, *FP* = false positive); *P* = discovery point (*C* = Compile Time, *R* = Runtime); *F* = bug type based on Essential Feature (*A* = Hang/Crash Error, *S* = Specification, *C* = Comparison, *H* = Hierarchy, *E* = EMI, ? = not further investigated); *Ver* = Latest Simulink version affected.

TSC	Summary	St	P	F	Ver
02382544	Simulink Block parameter specification mismatch (Constant)	NB	C	S	2015a
02382873	Internal rule cannot choose data-type (Add)	FP	C	?	2015a
02386732	Data-type support specification mismatch (PID Controller (2DOF))	NB	C	S	2015a
02472993	*Automated rate transition* failure (First-order hold)	NB	R	S, H	2017a
02476742	Block-reduction optimization does not work (Accelerator mode)	NB	R	E, H	2017a
02513701	Simulink hangs for large models with hierarchy	NB	C	A, H	2015a
02515280	Inconsistent result and ambiguous specification (SubSystemCount metric)	NB	C	S, H	2017a
02539150	Ambiguous results (selecting connection with multiple destinations)	NB	C	S	2017a
02565622	Limited support in Accelerator mode (First-order hold)	KB	R	C, H	2015a
02568029	timer does not execute callback as expected	FP	R	?	2015a
02614088	Undocumented specification (Variable Integer Delay)	KB	C	S	2017a
02705290	Incorrect data-type inheritance (multiple blocks)	NB	C	S	2017a
02869856	Incorrect specification for the Combinatorial Logic block	NB	C	S	2017a

Hang/Crash Error bug

When generating large hierarchical models, we noticed that Matlab's getAlgebraicLoops API hangs and makes the entire tool chain unresponsive (TSC-02513701).

Specification bug

After incorporating Simulink specifications into various SLforge phases, SLforge started to identify bugs caused by specification violation. For example, bug TSC-02472993 of Fig. 3.7 manifests when Simulink fails to handle blocks operating at different sample times,

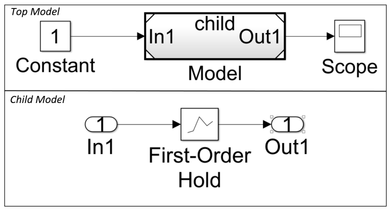

Figure 3.7: Bug TSC-02472993: The *Model* block (top) refers to the child model (bottom). Simulink fails to handle rate transition automatically, leading to a runtime failure.

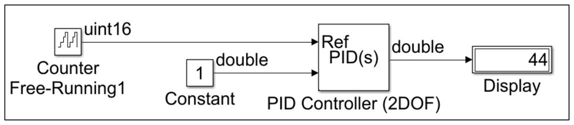

Figure 3.8: Bug TSC-02386732: While specified to only accept *double* inputs, Simulink does not raise a type error for this *PID Controller (2DOF)* accepting a *uint16*.

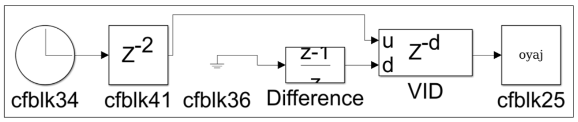

Figure 3.9: Bug TSC-02614088: In spite of supporting *double* data-type in its *d* port, block VID issued a data-type consistency as it *prefers integer* type.

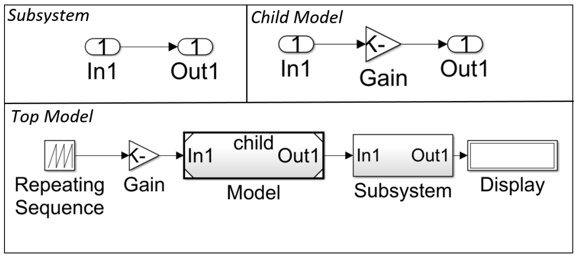

Figure 3.10: Bug TSC-02515280: For the child model (top right) Simulink's *Verification and Validation* toolbox API calculates inconsistent SubSystemCount values. MathWorks ruled the API specification ambiguous, as it did not properly define the API's scope.

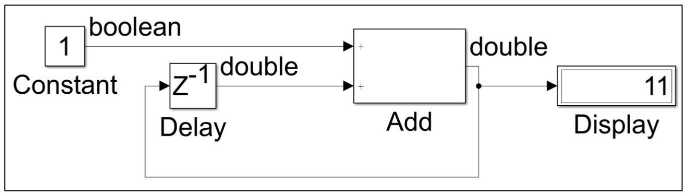

Figure 3.11: Issue TSC-02382873: When using the *internal rule* to detect the *Add* block's output data-type, the rule fails to choose a correct data-type for the second input port (e.g. *double*) and throws a compilation error.

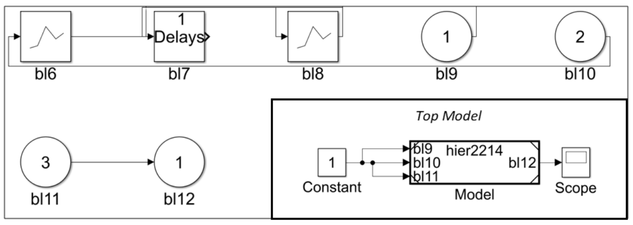

Figure 3.12: EMI bug TSC-02476742: The *Top Model*'s (in bottom right corner) *Model* block is a placeholder for the child model (top and left), where all blocks except *bl11* and *bl12* are dead.

leading to runtime failure which is not expected as per specification. The bug only occurs for the First-Order Hold block and when SLforge generates hierarchical models. As another example, Fig. 3.8 depicts Simulink's PID Controller (2DOF) block accepting data of type unsigned int, whereas the specification states that the block only accepts data of type double (TSC-02386732).

In another case we noted that in spite of supporting type *double* in port *d*, block *Variable Integer Delay* (block *VID* in Fig. 3.9) resulted in type-mismatch error. After reporting the issue, MathWorks suggested that the port "prefers" integer types and thus issued a type mismatch error when it was given a double type. This specification is not publicly available. Lastly, the Fig. 3.10 issue (TSC-02515280) MathWorks classified as expected behavior, where Simulink's count of the number of Subsystems did not match our count. However, part of Simulink's results are inconsistent and the specification has been found ambiguous, resulting in a new confirmed bug.

Comparison bug

In issue TSC-02565622, one Simulink instance could simulate the SLforge-generated hierarchical model in Normal mode but returned an error in Accelerator mode, due to inconsistent block sample rates. MathWorks confirmed this as a known issue that does not have a public bug report.

EMI bug

Fig. 3.12 illustrates Simulink bug TSC-02476742. Notice how only block bl11 is connected to an Outport block bl12, hence all remaining child blocks are dead and can be removed in EMI-testing. While EMI-testing in Normal mode removed all dead child nodes, EMI-testing in Accelerator mode failed to do so, which MathWorks classified as a Simulink bug.

3.6 Discussion

This paper performed the first large-scale study on publicly available Simulink models, to collect and observe various properties from them, which can be utilized to generate random models and serve as a collection of curated artifacts. However, some of the models are quite simple. We endeavored to classify such models in the Simple category manually, however, our approach may be imperfect and may suffer from human error. Opportunistically, we found complex and large models in our study and consequently, our collection of artifacts should be suitable for other empirical studies.

3.7 Related Work

Empirical studies of widely used programs date back at least to the 1970s [47] and have gained increasing interest due to the wide availability of open source programs [87]. For example, earlier work computed properties from Java programs [37, 19, 101] and used the properties to guide a random Java program generator [42].

Tempero et al. presented the *qualitas corpus*—a curated collection of Java programs [87]. Although similar work has been performed in other domains [12, 60, 84], we are not aware of related work in the CPS domain, which differs significantly from procedural or object-oriented languages.

Recent studies introduced measures for Simulink model modularity [25] and complexity [67], but only evaluated them on a limited number of models. In contrast we created a larger collection of 391 Simulink models. Similar to [87], our model collection may serve as a corpus for Simulink-model based empirical studies.

Recent work has found many compiler bugs using differential testing. To generate programs that are syntactically correct, many of the test generators harness the language's context-free grammar and a pre-determined probability table from which the generator chooses grammar elements [29, 41]. To generate programs that are also well-typed, McKeeman imposes the type information directly onto the *stochastic grammar* the generator uses [59].

Csmith, on the other hand, uses various analysis and runtime checks to generate programs with no undefined behavior [98]. Other techniques generate well-typed programs using knowledge of the type-system of the underlying language (e.g., JCrasher for Java [22]) and using constraint-logic programming (such as the Rust typechecker fuzzer [29]).

Whereas earlier approaches target compilers of textual languages (including procedural, object-oriented, and functional ones), they do not address the challenges inherent in testing CPS tool chains [16]. CyFuzz pioneered differential testing of CPS tool chains, but the prototype for Simulink was ineffective in finding new bugs. This work addresses CyFuzz's limitations by incorporating informal Simulink specifications in the random model generation process and generating larger models with rich language features, which led to finding new bugs.

Recent work complements randomized differential testing via EMI-testing. For example, Le et al. hammer C language compilers [48]. We harness the technique to create EMI-variants of Simulink models for the first time. Other work discusses the effectiveness of randomized differential testing and EMI for OpenCL compilers and performs a comprehensive empirical compiler testing evaluation [51, 11].

Among other works, Nguyen et al. present a runtime verification framework for CPS model analysis tools leveraging random hybrid automata generation [65, 44]. In contrast, our generator does not rely on model transformations [6], which may limit the efficiency of existing work [65]. Other testing schemes target parts of the CPS tool chain utilizing graph grammars [82, 81]. However, complete and updated formal specifications for most commercial CPS development tools are unavailable and such white-box testing in parts was found undesirable [76].

Sampath et al. discuss testing CPS model-processing tools using semantic *Stateflow* meta-models [76]. Unfortunately, the approach does not scale and updated specifications are unavailable, due to the rapid release cycles of commercial CPS tools [81, 29]. Fehér et al. model the data-type inferencing logic of Simulink blocks for reasoning and experimental purposes [32]. While these works focus on a small part of the entire CPS tool chain, we differentially-test the entire CPS tool chain harnessing the available informal (but updated) specifications.

SLforge is loosely related to test case generators for existing Simulink models [30, 9, 61, 34, 58, 79, 33] and verification and formal analysis of CPS models [78, 3, 102, 52, 46, 4]. But they do not aim at finding bugs in the Simulink tool chain.

3.8 Conclusions

This paper described the first large collection of public Simulink models and used the collected models' properties to guide random model generation. To further guide model generation we systematically collected semi-formal Simulink specifications. In our experiments on several hundred models, the resulting random Simulink model generator SLforge was more effective and efficient than the state-of-the-art tool CyFuzz. SLforge also found 9 new confirmed bugs in Simulink.

CHAPTER 4

A CURATED CORPUS OF SIMULINK MODELS FOR MODEL-BASED EMPIRICAL STUDIES[1]

Shafiul Azam Chowdhury

Lina Sera Varghese

Soumik Mohian

Taylor T. Johnson

Christoph Csallner

Shafiul Azam Chowdhury, Lina Sera Varghese, Soumik Mohian, Taylor T. Johnson, and Christoph 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 '18), pages 45-48. 2018

DOI: 10.1145/3196478.3196484

[1]Used with permission of the publisher, 2018

Abstract. Recent years have seen many empirical studies of model-based cyber-physical systems and commercial CPS development tool chains such as Matlab/Simulink. To benefit such research, this paper presents the by-far largest corpus of freely available Simulink models to date, containing over 1,000 models.

Surprising findings based on this corpus include that (a) tool support for metric collection is not adequate and (b) users do not reuse model components as they would in object-oriented programs. The paper both confirms and contradicts earlier findings that are based on significantly fewer models, suggesting the utility of the corpus for future research. While others have not yet leveraged this model corpus, we hope that our freely available corpus and infrastructure will benefit future model-based empirical research and tool development efforts, by reducing the model-collection overhead and thus easing evaluation.

4.1 Introduction

In model-based design of cyber-physical systems (CPS), engineers rapidly prototype their systems using graphical *models* in sophisticated development environments (e.g., Matlab/Simulink) [93]. Increased usage of such models across various industries (e.g., automotive, aerospace, and industrial automation) has elicited interest in understanding the model properties (e.g., size measures) and how they relate to quality attributes (e.g., complexity and comprehensibility) [67]. Many of these studies investigate structural model properties and propose new metrics based on the properties.

When evaluating various proposed metrics, most of the model-based studies only use a handful of models, which could adversely affect the evaluation. Besides, different studies compute measures using different sets of Simulink models, which makes their comparison problematic. Furthermore, studies often use proprietary models that are not publicly available, which makes reproducing results difficult.

A collection of publicly available models would facilitate evaluation and comparison of many model-based empirical studies. Besides, tools that operate on models (e.g., static analysis, refactoring, and clone detection tools) often require models of sufficiently large size and structural complexity, partly to evaluate scalability [27].

Building a sufficiently large collection of freely available Simulink models is thus vital and incurs non-trivial overhead. However, arbitrarily adding *any* model in such a collection may be undesirable. For example, studies may only be interested in complex models or in CPS domain-specific (e.g., automotive) models. Similar as for the corpus of Java programs [87], we investigate the various challenges in developing a *curated* collection of models (aka *corpus*).

Since such a model corpus is currently unavailable, insights into modeling practices are also unknown. This inspired the *SLforge* project to build the only-known large-scale collection of public Simulink models [17]. However, the main focus of this earlier work was testing the Simulink tool pipeline automatically.

SLforge collected 391 Simulink models and published their sources, but it did not focus on crafting a corpus. In contrast, we discuss the design challenges for creating a corpus and publish a much larger collection of 1,030 models, along with useful meta information, which would significantly reduce model-collection-overhead in future studies. Models in our corpus are large: 93 models have over 1k blocks, which is greater than the average number of blocks in the models used at Delphi, a large industrial Simulink user [52]. Furthermore, SLforge only studied the model metrics relevant to testing Simulink whereas we investigate interesting modeling practices as well as Simulink model complexity metrics. The corpus and the tools are freely available [5].

4.2 Background

This section provides necessary background information on model-based design using Simulink and next, the most related work.

4.2.1 CPS Data-flow Models and Simulink

Here, we briefly discuss modeling abstractions in Simulink. A graphical model (of a CPS) consists of *blocks*, which perform operations on their inputs and pass outputs to other blocks through *connection* lines. Simulink offers a variety of built-in blocks, organized in *libraries* and allows establishing *implicit* or hidden connections using From and Goto blocks [68]. To facilitate custom block-behavior Simulink avails placing native code (e.g.,

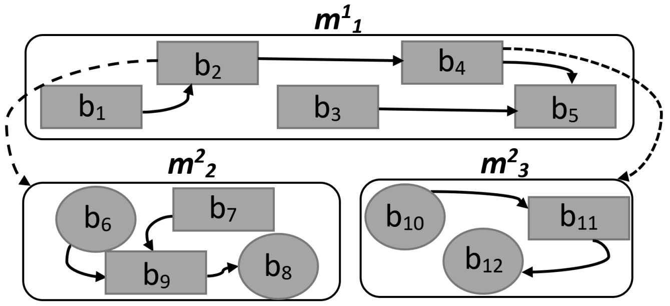

Figure 4.1: Example CPS model: Rounded rectangle = model; shaded = block; oval = I/O; solid arrow = dataflow; dashed arrow = hierarchy [17].

C) using the S-function interface. Besides *flat* models, Simulink offers hierarchical model creation using Subsystem and Model Reference features, which we collectively call *child-representing blocks*.

Fig. 4.1 contains an example hierarchical CPS model [17]. The parent to child model relation is acyclic. But within a model Simulink permits *feedback loops* (circular data flow). During compilation Simulink may reject a model if it fails to numerically solve feedback loops (aka *algebraic loops*) using solvers. Simulink offers different *simulation modes*. While in *Normal* mode Simulink "only" simulates blocks, it also emits some code for blocks in *Accelerator* mode. In-depth descriptions of Simulink modeling features are available [93].

4.2.2 SLforge

Crafting a curated collection of open-source programs is common in most programming domains [87]. However, the only large-scale study of public Simulink models we are aware of is the SLforge project [17]. While SLforge compiles a list of 391 publicly available Simulink models, its main focus is developing Simulink testing techniques. Whereas, we discuss corpus design challenges and publish the redistributable models in a single installation file, to ease replication and comparison of model-based empirical studies. Besides the (redistributable) models, our corpus includes *meta* information which empirical studies may find useful.

Next, unlike SLforge, we study model metrics relevant to analyzing complexity and modeling practices in general, based on the largest collection of 1,030 publicly available models. While SLforge investigated metrics (i.e., the number of blocks and connections and maximum hierarchy depth in a model and library-usage information) relevant to automated testing, we define and investigate metrics mostly to explore model complexity. Based on our metrics data, we further perform a lightweight comparison with other empirical studies and discuss interesting findings. Furthermore, we extend SLforge's tools to support the new metrics.

4.3 Corpus Design Choices

To identify the corpus contents we have used the following criteria, which we expect to be useful for many model-based studies.

CPS Domain

To support various domain-specific studies, we identified the qualitative attribute *CPS domain* (e.g., Automotive or Avionics) whenever possible for each project, from author-provided descriptions and "tags".

Trivial Models

Some studies filter out example, toy Simulink models and examine them separately [17]. We included the manually-identified "trivial" models in the *Simple* group (Section 4.4.1).

Choice of Projects

Extending SLforge's model collection [17], we included 96 additional projects (each containing one or more models) from the Matlab Central repository, filtering by highest download count, and 12 projects from the SourceForge public repository.

Content Type

We only include a project in the corpus if it releases models in the mdl or slx formats since these formats are most widely accepted by both engineers and analysis tools. Additionally, when projects distribute code and generated executables we include them as well since studies may choose to analyze them.

Test Harnesses and Libraries

Many projects come with test harnesses and custom *libraries*, which may themselves be Simulink models. Since studies may choose to analyze them separately, we identified and published their list.

Toolbox Requirements

We extract and include the (mandatory and optional) toolbox requirements information whenever available from the project websites.

4.4 A Study of Model Metrics

Here, we define interesting model metrics and utilizing the corpus, investigate metrics related to complexity and modeling practices. We discuss findings and compare with earlier work.

4.4.1 Model Groups

Similar to SLforge [17], we group models into four groups: (1) *Tutorial* (*t*)—the Simulink proprietary models ; (2) *Simple* (*s*)—the models we manually filtered out as toy-example ones; (3) *Advanced* (*a*)—the non-trivial models; (4) *Other* (*o*)—the models we were not able to manually study for time reasons.

4.4.2 Model Metrics

Mostly using min-max whisker box-plots, we discuss the following model metrics. We have not considered the custom library files (Section 4.3) as CPS models in this study.

Table 4.1: Overview of the collected models: Total number of models (M), models we could compile readily—without installing additional toolboxes (Cm), hierarchical models (H), number of blocks (B), non-hidden connections (C), and all connections (Ch). We could not compute solver (Fixed-step (Fxd) and Variable-step (Var)) and simulation mode (Normal (N), External (E), PIL (P), and Accelerator (A)) information for all of the models.

		SLforge					Our Corpus											
	Group	M	Cm	H	B	C	M	Cm	H	B	C	Ch	Solver		Simulation Mode			
													Fxd	Var	N	E	P	A
t	Tutorial	41	40	40	10,926	11,541	41	40	40	10,926	11,541	11,828	13	28	41	0	0	0
s	Simple	156	99	136	7,187	7,121	442	208	325	14,203	14,013	14,261	210	198	389	18	1	0
a	Advanced	167	66	165	118,632	116,608	452	147	347	406,185	403,503	429,033	122	227	344	0	0	5
o	Other	28	14	21	8,317	9,577	136	29	124	13,117	14,379	14,600	111	24	80	53	0	2
	Total	391	219	362	145,062	144,847	1,071	424	836	444,431	443,436	469,722	456	477	854	71	1	7

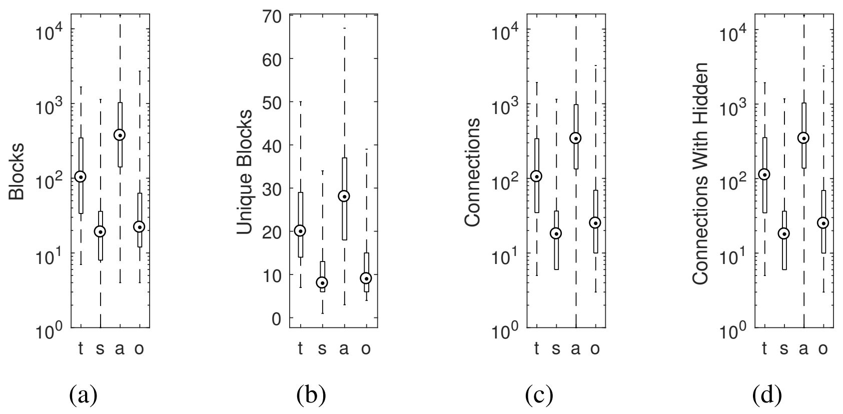

Figure 4.2: Model metrics: Total blocks (a), unique blocks (b), connections excluding hidden (c), and including hidden (d).

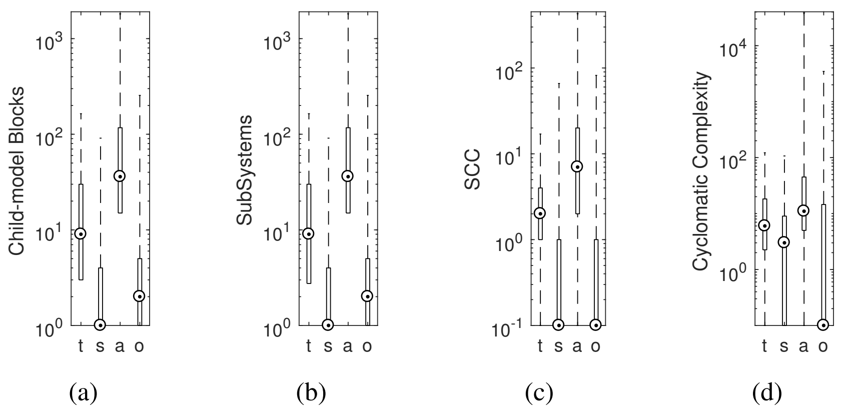

Figure 4.3: Metric results: Number of (a) child-representing blocks, (b) contained subsystems, and (c) strongly-connected components; (d) cyclomatic complexity.

Table 4.2: Most frequently used blocks (besides the top-3 Inport, Outports, and SubSystem), in descending order.

	Blocks
t	Product, Constant, Sum, Gain, From, Selector, Mux, Demux, Terminator, Goto, UnitConversion, BusSelector, Fcn, Integrator, Math, Trigonom.
s	Constant, Gain, S_Fun, Terminator, Sum, DataTypeConv., Demux, Mux, Scope, PMIOPort, From, Product, RelationalOp., Goto, Ground, Integr.
a	Constant, From, PMIOPort, Sum, Gain, Goto, Product, Mux, Demux, RelationalOp., Switch, Fcn, SimscapeMulti., PMComp., Ground, Terminator
o	Constant, Gain, Sum, Product, Termin., Mux, S_Fun, Delay, RelationalOp., Demux, From, ZeroOrderHold, DataTypeConv., Integr., Saturate

Blocks and Connections

The current CPS literature heavily uses the number of blocks and connections, but many studies do not specify if they include "hidden" blocks (those in Masked subsystems) and implicit (aka hidden) connections. We include hidden blocks and give counts for both hidden and regular connections (Fig. 4.2). A related source of confusion is that two publicly available tools (sldiagnostics and the tool in [88]) report different block counts (e.g. 5,700 vs. 10,953) for the same model, as the second tool multiplies the number of blocks in a referenced model by the number of time that model is re-used [88]. The second tool captures the "net functionality" represented by a model, so we use it for our counts.

Simulation Complexity

We investigated whether the number of strongly-connected components in the graph representation of a model captures its complexity in terms of numerically simulating it (Fig. 4.3c). While algebraic loops also incur simulation complexity, we found that only 18 models have such loops.

Child-model Representing Blocks

We count child-representing blocks (Fig. 4.3a) and the number of contained subsystems (NCS) (i.e., the number of blocks in that subsystem) (Fig. 4.3b), as earlier work relates the latter to model complexity [67]. Interestingly, the distribution of the two metrics is almost identical, implying that model-referencing, which is considered as good modeling practice in general, is not widely used in the corpus-models [93].

Hierarchical Modeling

We found that the median number of blocks in a particular hierarchy level does not exceed 17—an observation similar to the "small class phenomenon" in object-oriented (OO) programs [101], where some 57% of the studied Java classes are smaller than 65 LOC on average. One drawback of the small class phenomenon in OO is that much of the conceptual complexity of understanding an OO program (vs. traditional procedural ones) is now in the inter-procedural call and override relationships (vs. the traditional intra-procedural control and data flow within each large method body). In Simulink, we note that most of the hierarchies are incurred by the subsystems, which are not reused or “subclassed”. Consequently, this drawback of the small-class phenomenon in OO may not apply to Simulink.

Child-model Reuse

Simulink allows reusing some Referenced Model in multiple places in the same model, which in OO may be similar to multiple instantiations of a class. However, our study found only one model using this feature, in the Tutorial group. So, at least from our model sample, it appears as if this feature is not as widely used as multiple object instantiation in OO languages. To reuse functionality, do Simulink users instead reuse S-functions? We found very low ($< 0.5\%$ median value) S-function reuse rate across all model groups.

MathWorks Cyclomatic Complexity

MathWorks defines an object's cyclomatic complexity (e.g., of a block) as $\sum_{i=1}^{n}(o_i - 1)$, where n is the number of the object's decision points and o_i is the number of possible outcomes at the i^{th} decision point [93]. We adopt the definition as it is widely used [67] (Fig. 4.3d), further noting that the MathWorks tool cannot compute cyclomatic complexity for non-compilable models. Consequently, we could not use the tool for a large portion of the models in the corpus.

Most-used Blocks

In Table 4.2, we examine the most frequently used 15 blocks, noting that Advanced models more frequently use blocks from the Simscape toolbox (e.g., PMIO and SimscapeBlock), which enables rapid creation of complex physical systems [93].

Simulation Configuration

Although these metrics do not relate to complexity, we explore the usage of solvers and simulation modes in Table 4.1 as these are major configuration options [17].

Others

We compute the number of unique blocks (distinct block-types) in a model and compilation time, primarily to explore whether these metrics correlate with cyclomatic complexity.

4.4.3 Replicating Earlier Model-based Studies

To compare with the findings of a recent model-based study by Olszewska et al., we conduct a pairwise correlation analysis on the metrics, namely cyclomatic complexity, compilation time, number of blocks and connections, number of child-representing blocks and NCS, number of strongly connected components and maximum hierarchy depth [67]. We compute pairwise Kendall's τ mainly to compare with the other study. All the metrics are positively correlated to each other (0.05 significance level).

From our observation, cyclomatic complexity is mostly correlated with the maximum hierarchy depth in a model (0.5509) and the NCS metric (0.5297). In contrast, Olszewska et al. identified NCS as mostly correlated, however, they do not compute correlation with hierarchy depth count and used a single Simulink model in their study whereas we used our full model collection (minus those for which we could not collect all metrics and Simple models) models (listed in [5]).

Other observations diverge from earlier work. For example, earlier work found Matlab Central models to have ten times fewer blocks than industrial models (the latter had some 752 blocks on average) [52]. However, our current collection contains much larger Matlab Central models.

4.5 Conclusions

In this work, we present the largest corpus of freely available Simulink models to date. Using the corpus, we explore interesting modeling and complexity metrics. Previously unknown findings in modeling practices and a lightweight evaluation of earlier model complexity study suggest the utility of the corpus in future model-based studies, by reducing the model-collection overhead and supporting evaluation and comparison of such studies. We will endeavor to grow the corpus and investigate metrics to capture model complexity utilizing our infrastructure.

CHAPTER 5

SLEMI: EQUIVALENCE MODULO INPUT (EMI) BASED MUTATION OF CPS MODELS FOR FINDING COMPILER BUGS IN SIMULINK

Shafiul Azam Chowdhury

Sohil Lal Shrestha

Taylor T. Johnson

Christoph Csallner

Abstract. Finding bugs in commercial cyber-physical system development tools such as MathWorks Simulink is important in practice, as these tools are widely used to generate embedded code that gets deployed in safety-critical applications such as cars and planes. Equivalence Modulo Input (EMI) based mutation is a new twist on differential testing that promises lower use of computational resources and has already been successful at finding bugs in compilers for procedural languages. To provide EMI-based mutation for differential testing of cyber-physical system development tools, this paper develops several novel mutation techniques. These techniques deal with CPS language features that are not found in procedural languages, such as an explicit notion of execution time and zombie code, which combines properties of live and dead procedural code. In our experiments the most closely related work SLforge found two bugs in the Simulink tool versions R2017a and R2018a. In comparison, SLEMI found a super-set of issues, including 10 confirmed as unique bugs by MathWorks Support.

5.1 Introduction

Commercial cyber-physical system (CPS) development tools are complex software systems that may contain bugs. Finding such bugs is hard as the CPS development tools do not have complete formal specifications [82, 29, 39, 8] and the tools' source code is not available either. While state-of-the-art CPS tool bug-finding approaches such as SLforge [17] have had some initial success, they also have two key limitations, i.e., they are (a) fundamentally slow and (b) limited to synthetic CPS models.

Finding bugs in commercial cyber-physical system development tools efficiently is important however, as the correctness of CPS development tools is crucial in practice. For example, the CPS development tool MathWorks Simulink [90] is a de-facto industry standard in several safety-critical domains, including automotive, aerospace, and health care [99]. Engineers widely use Simulink to generate embedded code from CPS models and deploy the generated code in safety-critical applications [9, 68, 38]. So a Simulink tool bug may lead to compile errors or inject subtle unexpected behaviors into safety-critical applications, e.g., in cars or planes [11].

To side-step the lack of CPS tool specifications, state-of-the-art CPS tool bug-finding approaches such as SLforge perform differential testing [59, 77, 98] on the CPS tool. By invoking two configurations of the same CPS tool on the same generated model, SLforge may trigger a *CPS tool bug* if the two configurations yield different results. But generating a valid synthetic CPS model is computationally expensive. For example, due to incomplete CPS language rules SLforge may require several "feedback-directed" iterations to automatically fix remaining Simulink compile errors.

A recent differential testing twist promises to address these limitations. Mutating a given program in a way that preserves its execution semantics on the given inputs has proven effective in finding bugs in C compilers [48]. These *Equivalence Modulo Input* (EMI) based mutation schemes perform a small program modification, which may be computationally cheaper than generating a program from scratch. Besides speeding up program generation and enabling the use of existing models, EMI-based mutation also enables finding compiler bugs by comparing two identically configured compiler executions (on equivalent input programs).

Recent work has laid the groundwork for evaluating CPS mutation approaches. Evaluating such approaches requires input CPS models and traditionally such models were not easy to obtain in sufficient numbers. For example, in the case of Simulink, random Simulink model generators and a corpus of public Simulink models only became available recently [17, 18, 16].

While prior work has reported promising results on EMI-based mutation for finding bugs in C compilers [48, 49, 85], it is not straight-forward to apply the existing EMI-based mutation schemes to CPS tools. As a case in point, the only existing approach described as EMI-based mutation for CPS models (SLforge) does not really follow the equivalence modulo input paradigm. Instead of preserving behavior for a single given input, SLforge performs a static mutation that preserves equivalence for all possible inputs (by using the Simulink compiler to remove all dead code). While the resulting mutation is also EMI (since maintaining equivalence on all inputs implies maintaining equivalence on the given input), it is a severely restricted approach and in the case of SLforge only yields one mutant per input model.

Specifically, SLforge does not leverage model runtime data, does not delete or modify any code the compiler does not remove, and does not insert any code. Given these limitations, it is perhaps not surprising that SLforge's "EMI" component has only found one bug.

The core challenge of EMI-based mutation of CPS models is that CPS languages are quite different from procedural languages, e.g., CPS languages have an explicit notion of execution time. CPS languages also have a different notion of when code is dead, which gives raise to zombie blocks (Section 5.2.2) in CPS models that do not exist in procedural code.

To address these challenges, this paper reviews several key differences between CPS models and procedural code. Specifically, we describe novel techniques for mutating CPS models that use an explicit notion of execution time and zombie code, i.e., code that has properties of both dead and live procedural code. We implement these techniques in the new SLEMI tool and empirically evaluate their effectiveness for finding bugs in the Simulink tool. SLforge, the state-of-the-art approach for finding bugs in the Simulink tool via EMI has found one bug. In contrast, SLEMI has to date found 10 unique issues confirmed as bugs by MathWorks Support in Simulink versions R2017a and R2018a. To summarize, this paper makes the following major contributions.

- The paper describes novel techniques for EMI-based mutation of CPS models, including techniques for dealing with language features that do not exist in procedural languages.

- Via the novel SLEMI tool, these techniques found 10 confirmed bugs in the widely used CPS development tool Simulink.

- Within set time and computational resources, SLEMI found more Simulink bugs than its closest competitor SLforge.

- All SLEMI code and evaluation data are open source[1].

[1]Currently anonymous for double-blind review: https://github.com/icse2020-emi/slemi

5.2 Background

This section contains necessary background information on key features of CPS modeling languages, how their data propagation, control flow, and "dead code" notions differ from procedural programming, our resulting notion of zombie code, and state-of-the-art approaches for finding bugs via EMI-based mutation and differential testing.

5.2.1 Block Diagrams and CPS Tool Chains

While in-depth descriptions of CPS languages are available elsewhere [62, 72, 93, 68, 17], following are the key concepts. In a cyber-physical system (CPS) development tool (e.g., Simulink), a user designs a CPS as a diagram or model m that consists of blocks and their connections. A block accepts data through its *input ports*, typically performs on the data some operation defined by a discrete or continuous function, and may pass output through its *output ports* to other blocks, along (directed) *connection* edges. More formally, each connection $c \in m.C$ is a tuple $\langle b_s, p_s, b_t, p_t \rangle$ of source block b_s, its out-port p_s, target block b_t, and its in-port p_t [17].

Since a typical CPS tool supports a wide range of modelling styles we do not further detail the connection semantics here. For example, in a tool's dataflow semantics a connection c_1 takes its source block's output data d_1 and eventually delivers d_1 to c_1's target block. However a CPS tool may at the same time support other semantics, in which, for example, a source block may overwrite data d_2 on a connection c_2 before c_2's target block can read d_2.

A model m typically acquires its inputs from sensors, whose values it samples at a user-defined frequency (e.g., 10 times per second). Each sample yields a new input vector i (containing one value per sensor) that the model processes in the execution order defined by the model's connection edges [62]. To affect its environment, a model typically has a set of *output blocks* (or sinks) m_{out} such as Fig. 5.1's $Out1$ and $Out2$ blocks, which can emit model output values to a display, another model, or a hardware actuator.

Commercial CPS tools specify the *datatypes* each port of each block supports (e.g., "either double or uint32"). If the user does not explicitly configure a port's datatype, then

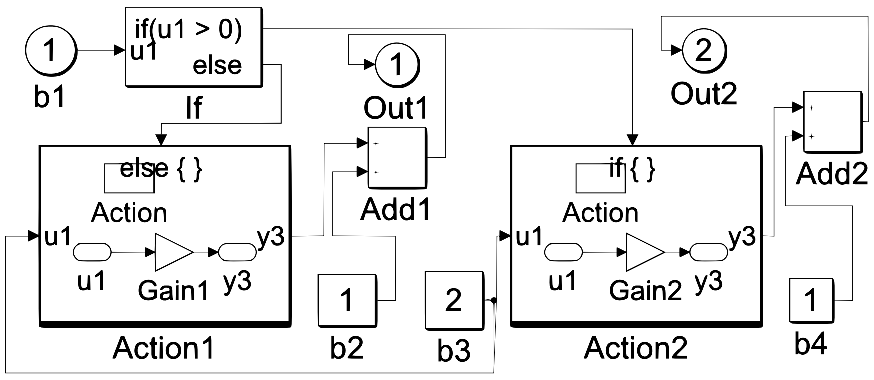

Figure 5.1: Example valid Simulink model: While $Action1$ is on a false-branch when $b1$ receives non-zero positive input, $Action1$'s values can still affect the outside world, making $Action1$ a zombie block.

the CPS tool infers and assigns a concrete datatype (e.g., "double"). If the user has under-constrained the blocks' datatypes, the tool may heuristically break "best datatype match" ties. We define the $T_{dt}(b, p)$ function to retrieve the resulting datatype of a given block b's port p.

When creating a model, users can re-use and customize standard blocks from built-in *libraries* supplied by the tool chain. Most blocks have user-configured parameters that may affect the block's output values. The user may also create blocks from scratch via segments of procedural code (e.g., in C or MATLAB).

Commercial CPS tool chains such as Simulink and LabVIEW do not have a specification that is complete, formal, and up to date. Besides partial informal descriptions, tool chain semantics are only defined via their code base [82]. Let $valid_{\mathcal{L}}(m)$ indicate if tool chain $\mathcal{L}$ accepts model m, i.e., compiles m without error. As an example, for a model to be valid, Simulink must be able to infer consistent datatypes for each port left unspecified by the user. Simulink uses several heuristic rules to infer and propagate datatypes, e.g., forward, backward, and block-internal propagation [32, 91].

After compilation users simulate models, 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. Commercial tool chains typically offer different simulation modes. For example, Simulink *Normal* mode "only" simulates blocks, *Accelerator* mode speeds up simulation by emitting native code, and *Rapid Accel-*

erator mode produces a standalone executable.

Besides flat models, CPS development tools offer hierarchical models (e.g., via Simulink's $\mathrm{Subsystem}$ and $\mathrm{Model\ Referencing}$), where the parent to child model relation is acyclic. A tool chain may permit a loop in the model's connection relation (aka a *feedback loop*) if it can numerically solve it.

5.2.2 Zombies: Output Data From a False Branch

How to best deal with conditional execution in block diagrams has been an open research question for decades, e.g., in the dataflow literature [45]. While well-understood in procedural programming, conditional execution differs significantly in block diagrams, which complicates the dead vs. live code distinction and therefore EMI.

For example, while impossible in procedural code, the (valid) Simulink model of Fig. 5.1 simultaneously returns values from both its true and false if-then-else branches. Assume that in the current model execution the user provides via block $b1$ a constant value of 5 as input to the model and thus to the control-flow conditional If. Given this input, in all simulation steps of this execution the conditional will thus select $Action1$ as the false-branch. Unlike in procedural programming, this false-branch still has a user-configured (here zero) output value. The subsequent increment causes $Out1$ to constantly emit 1, which can affect the outside world.

In a procedural setting $Action1$ would be dynamically dead code and we could delete it for this execution trace. But in our block diagram setting $Action1$ is not dead. Instead, a block b is *dead* and can be removed if there is *no path* from b to any output block (and neither b nor its successor blocks can produce other side-effects). Simulink has built-in tools to remove such dead blocks. Unlike in procedural programming, this "no path exists" notion does not depend on runtime data, so for block diagrams we do not distinguish between statically and dynamically dead.

A block may *never be activated*, e.g., because it is on an always-false branch such as $Action1$. We call such a block a *zombie* (as in live-dead hybrid), as it has properties of both procedural live code (it has program values) and procedural dead code (no computations take place). A *static zombie* is a zombie in all possible model executions. A *dynamic* *zombie* is a zombie in the current model execution (e.g., $Action1$). Neither Simulink nor external tools we are aware of detect or minimize dynamic zombie blocks.

$Action1$ is a *top-level zombie*, its (default) value can reach the outside world. In contrast, a *nested zombie* such as Fig. 5.1's $Gain1$ block is nested inside a top-level zombie. A nested zombie cannot influence the outside world, as its top-level zombie never processes the nested zombie's (default) value. A nested zombie is thus conceptually similar to procedural dead code.

Finally, a block is live if it has both a path to an output block (or another side-effect) and gets activated. A *dynamically live* block is live in the current model execution. It may be a zombie (but not dead) during other executions. A *statically live* block is live in all possible executions.

5.2.3 Differential Testing, EMI, And SLforge

Differential compiler (or CPS tool chain) testing compares two execution traces that compile and execute a program (or model). By design these two traces are supposed to be equivalent, i.e., produce the same result values. If the results differ we have likely found a compiler bug. More formally, for programs m and n, and program parameters p and q, based on our understanding of the programming language semantics $[\![\cdot]\!]$, we expect equal execution results, i.e., $[\![m(p)]\!] = [\![n(q)]\!]$. We expect to have found a bug if two compiler configurations C and D for this language instead produce different results, i.e., $C(m)(p) \neq D(n)(q)$.

One way to instantiate this framework is to fix a program plus parameter combination ($m = n$, $p = q$) and only differ the tool configuration ($C \neq D$). Indeed, well-known differential testing approaches such as Csmith have found many compiler bugs by running randomly[2] generated programs under varying compiler configurations (i.e., $C(m)(p) \neq D(m)(p)$). In the CPS world, existing approaches for Simulink similarly have varied compiler optimization levels, numerical solvers, simulation modes, and code generators [20, 16, 17].

Equivalence modulo input (EMI)-based differential testing, on the other hand, has typ-

[2]As common in the literature, in this paper by "random" we mean "pseudo-random".

ically instantiated this framework by fixing a tool configuration plus program parameter combination ($C = D$, $p = q$). In other words, EMI-based approaches use a program m and one of its mutants n that is expected to be functionally equivalent to m on the given input p, i.e., $m \neq n$ and $[\![m(p)]\!] = [\![n(p)]\!]$. Again, different results suggest a compiler bug (i.e., $C(m)(p) \neq C(n)(p)$). While there has been recent interest in EMI for testing C compilers [48, 49, 85, 83], besides SLforge we are not aware of EMI-based testing work for block diagram or CPS model languages.

Random program generators have significantly improved the bug-finding capability of both types of differential testing instantiations [48, 11, 98]. Existing random Simulink model generators include CyFuzz [16] and its successor SLforge [17]. Besides comparing traces of different tool configurations, SLforge also performs a very restricted form of EMI-based mutation and is thus the approach most closely related to SLEMI. Specifically, after generating a random valid Simulink model, SLforge optionally runs Simulink's static *block reduction* tool to delete all dead blocks.

5.3 SLEMI: CPS Tool Chain Testing via EMI

Existing EMI-based compiler testing approaches focus on procedural programs [48, 49, 85]. CPS models are different in several ways, e.g., they may emit output from several parts of the model at the same time. They are also typically simulated over a finite number of simulation steps, where at each step s the model consumes a separate input vector i, amounting to a sequence I of input vectors.

We adapt the framework of Section 5.2.3 to CPS models. To keep our definition simple, we represent block b of model m at simulation step s (after m has processed s input vectors from input vector sequence I) as $m(I)^s.b$. Since commercial CPS tool chains support floating-point datatypes, we compare block outputs via a tolerance.

In other words, we consider x and y equivalent (i.e., $x \approx y$) if $|x - y| < \epsilon$, where ϵ is configurable (10^{-16} by default) [16]. We thus consider two CPS models m and n (n obtained by mutating m, i.e., $n = m'$) equivalent modulo a common sequence I of input vectors, i.e., $m \equiv_I nI$, if both models are valid, have the same output blocks, and the CPS tool chain semantics $[\![\cdot]\!]$ at all time steps s prescribes equivalent values for all blocks b that

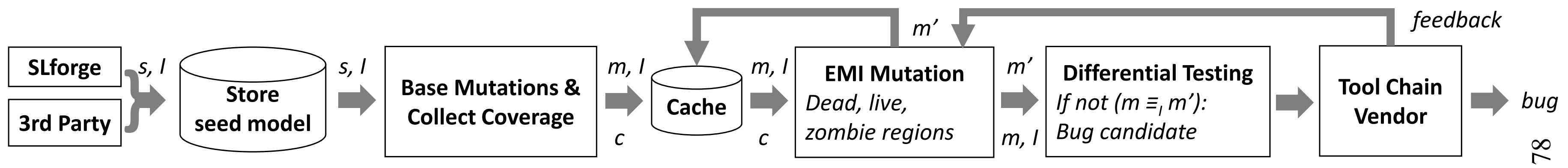

Figure 5.2: Overview: SLEMI first obtains seed model s with input vector I from a real-world corpus or a random generator (e.g., SLforge), performs one-time base mutations to yield model m, and collects m's coverage c (on I). An EMI-based mutation then yields a valid equivalent (on I) model m' for finding tool chain bugs via differential testing.

are common to both models, as follows.

$$m \equiv_I nI \iff valid(m) \wedge valid(n) \wedge (m_{out} = n_{out}) \wedge \\ \forall \langle b \in (m \cap n), s\rangle : [\![m(I)^s.b]\!] \approx [\![n(I)^s.b]\!]$$

Fig. 5.2 outlines our approach. SLEMI takes as input real-world and randomly generated CPS models together with their input values. We first filter out invalid models (as they are not suitable for differential testing) and then execute each seed model on its inputs to collect block-level coverage information (on all model hierarchy levels, via *Simulink Coverage* [48, 92]). Then SLEMI performs several one-time base mutations (Listing 5.1) and stores data in a persistent cache. We then mutate a model by removing and adding blocks.

Given the lack of a full formal specification of Simulink, we had to revert to an iterative approach for developing mutation operations that maintain equivalence modulo input. In other words, we expect that each of our mutations converts a model m into m' such that $m \equiv_I m'I$ holds. If via differential testing SLEMI determines that $m \equiv_I m'I$ did not hold, we report the issue to MathWorks. MathWorks confirming a bug increases our confidence in the mutation's EMI property. A feedback of "false positive" tells us our mutation was not EMI, gives us a better understanding of the tool's (otherwise undocumented) semantics, and we have to adapt (or abandon) the mutation operation accordingly.

Compared to model simplifications performed by optimizing compilers (including profile-guided optimizers, trace compilers, etc.) our below model mutation strategies are more general. Optimizing compilers rewrite programs toward a concrete goal (such as increasing execution speed or minimizing power consumption). For example, while optimizing compilers would not consider adding complex extra execution logic into live or dead code, we are interested in all EMI mutations, in our bid to find additional CPS tool-chain bugs.

5.3.1 Base Mutations: Annotate Seed Models

SLEMI's base mutations deal with two challenges that did not occur in earlier work on EMI-based differential testing [48, 49, 85], i.e., datatype inference and sample time infer-

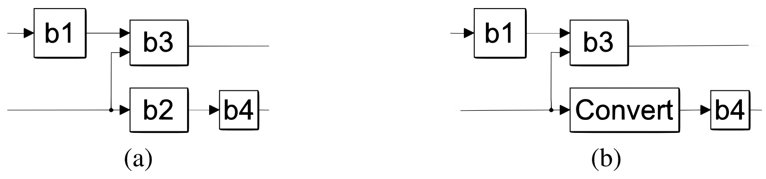

Figure 5.3: Example seed model excerpt without explicit type specifications (a) where Simulink propagates *double* via $b2$. Replacing nested zombie $b2$ with a $Convert$ $\mathrm{Data\ Type\ Conversion}$ block (b) may yield different type inference results.

ence.

Annotating Seed Models With Port Datatypes

The first challenge is introduced due to datatype inference. Instead of enforcing datatypes to be fully specified on every single port on each block, which can get cumbersome on large-scale models, Simulink infers unspecified datatypes based on data dependencies and optional partial specifications. For models with under-constrained datatypes, even a small SLEMI-induced mutation that may intuitively seem like it should be EMI can trigger vastly different inferred datatypes, which could create false warnings during differential testing.

Good examples of this problem are model regions that are dynamically zombie. Simulink's datatype propagation rules may rely on these zombie regions and mutating them may severely affect the datatypes Simulink infers in the surrounding regions. To give SLEMI more EMI mutation choices, we therefore first want to annotate the seed model with the datatypes Simulink infers. The seed's types thus remain available for compilation even after extensive mutations that may otherwise alter Simulink's datatype inference results.

As a concrete instance of this challenge, in the Fig. 5.3a child model excerpt of a larger seed model (omitted for brevity) Simulink propagates type *double* from block $b2$ to blocks $b3$, and $b1$. When we replace the nested zombie $b2$ with a *TypeCast* block ($\mathrm{Data\ Type\ Conversion}$ in Simulink) yielding Fig. 5.3b, then Simulink counter-intuitively propagates *int* to $b3$ and $b1$, which is not compatible with $b1$, yielding a compile error.

Listing 5.1: Base mutations to preprocess seed m using $set(b, p, t)$, which fixes out-port p's datatype to t. Besides changing to a fixed-step solver (Section 5.3.1), base mutations are EMI.

```
preprocess($m$, $I$) // returns $m$
  change to fixed−step solver // not EMI
  execute $m$ using input $I$
  for each block $b \in m$ : // collect inferred properties
    collect execution coverage
    collect inferred datatype and sample time
    annotate sample time if $b$ is a Source block
  for each connection $c \in m$ : // add types
    set($c.b_s$, $c.p_s$, $T_{dt}(c.b_s, c.p_s)$) // source out−port
    $d$ := new $TypeCast$ block // for target in−port
    set($d$, 0, $T_{dt}(c.b_t, c.p_t)$) // $d$'s only out−port (0)
    connect $c.b_s \rightarrow d$ and $d \rightarrow c.b_t$ // rewire
```

Inferring the datatypes of all seed model blocks and explicitly specifying these types in the model is an EMI mutation, as it does not change the outcome of model compilation or subsequent simulation. Simulink offers two options for specifying types and SLEMI uses both. First, some (but not all[3]) blocks have a parameter that sets the block's out-port types. Second, TypeCast blocks (such as Data Type Conversion) have a defined output type. Placing one of them before a block b lets one define b's in-port type.

The lower half of the Listing 5.1 pseudo code summarizes these type annotation steps. First, for each block b_s that has this option, SLEMI sets the block's output type to the Simulink inferred type. Second, SLEMI adds a fresh TypeCast block d before each block b_t, to annotate b_t's input type.

Even combining both annotation strategies does not fully specify all types, as the out-port parameters do not cover all blocks and TypeCast leaves its in-port unspecified. However, this combination has been sufficient and did not create any false warnings in our experiments.

[3]For example, Simulink's Discrete Transfer Function block does not support specifying *double* datatype at its out-port via block parameters. One can achieve this effect only by controlling the block's in-port datatype.

Dealing With Sample Time Inference

The second challenge not found in earlier EMI-based testing for procedural languages is that in a CPS model each block has a sample time. At a high level, this challenge is similar to the previous datatype inference issue. As a concrete example, commercial CPS tool Simulink encourages the user to specify the sample time only for a subset of the blocks and then let the tool infer the sample time for the remaining blocks (using forward and backward propagation [55]). Again, a small mutation that at first glance seems like it may be EMI can trigger the CPS tool to infer vastly different sampling times for large portions of the model, which in turn yields different model outputs, yielding a false bug warning.

As an extreme example, some blocks directly expose their block-specific sampling frequency, such as Counter Free-Running, which plainly returns the number of times it has been sampled as its output [93]. The tool chain inferring different sample times for such a block in a seed and a mutant model yields different results.

Similar to datatype inference, before we mutate the seed model we want to preserve the sample time inference results from the seed model, to give SLEMI more options for EMI mutations. However, different from the datatype issue, attempting to annotate each block proved to be a dead end. The Simulink documentation encourages users to only annotate either the source blocks or the sink blocks [56] and our initial bug reports that contained blanket sample time annotations for all blocks were rejected for that reason. Based on this feedback, SLEMI now only adds inferred sample time annotations to the seed model's source blocks.

A key feature of CPS development tools is their support for simulation of continuous-time models via variable-step numerical integration [54]. In other words, at each simulation step the tool's output includes the next simulation step's length (aka the next simulated execution time point). By varying time steps, the tool may thereby improve simulation precision. From SLEMI's perspective, this again may cause a seemingly small mutation to trigger non-EMI model changes.

To side-step this and related continuous-time issue, SLEMI currently performs one base mutation that is not EMI. In the Listing 5.1 pseudo code this mutation appears as the first step of switching the seed model from a variable-step to a fixed-step solver, the latter being widely used in practice [54, 18]. For such models SLEMI also disables the related zero-crossing detection feature. While it restricts SLEMI's bug search space, this non-EMI base mutation does not impact the correctness of the overall workflow, since for differential testing SLEMI only uses preprocessed models.

5.3.2 Mutating Nested Zombie Regions

At a high level this mutation is similar to mutating dead code regions in procedural languages [48]. Since a top-level zombie ignores any values (including defaults) coming from its nested zombie region, this mutation can change the nested zombie region freely, without being observable from the outside world.

Due to preprocessing, this mutation is easy to implement, as during preprocessing SLEMI has added extensive TypeCast nodes throughout the model. This means that even removing a random block within a nested zombie region only has a relatively small chance of introducing a compile error. This contrasts with procedural languages, where, for example, removing a local variable definition likely causes a compile error in subsequent variable use. So while existing EMI tools tend to remove an entire dead region, SLEMI removes individual blocks from nested zombie regions.

With the per-block coverage from preprocessing SLEMI identifies nested dynamic zombies. For example, in Fig. 5.1 block $Gain1$ is a nested zombie, as it is nested inside the top-level zombie $Action1$. After deleting a nested zombie block, SLEMI randomly reconnects its predecessor blocks to its successor blocks (top of Listing 5.2). This may leave some of the deleted zombie's predecessor blocks unconnected, when the number of incoming connections (from predecessors) to the deleted zombie block is greater than the number of outgoing connections (to successors). By default SLEMI ensures that such unconnected predecessor blocks do not get treated as dead code, as Simulink may otherwise remove them.

Listing 5.2: Create EMI mutant from preprocessed m.

```
mutate($m$, $I$)
```

```
if randomly chosen block $b \in m$ is nested zombie:
  delete $b$
  randomly wire all of $b$'s successors to $b$'s predecessors
else if $b$ is top−level zombie:
  replace $b$ by $d$ such that $d$ mimicks $b$'s output
  ensure $d$ has $b$'s sample time
else: mutate live hierarchy or fork new connection
```

5.3.3 Mutating Zombie And Live Regions

Since prior work has found mutating live code more effective than mutating dead code [85], we adapt live mutations to CPS models and generalize them to also cover top-level zombie blocks. Compared to the earlier work's mutation based on program synthesis [49, 85], SLEMI currently focuses on the mutations of the lower part of Listing 5.2.

First, SLEMI's *live-hierarchy extraction* mutation extracts a live region and promotes it to its own child model (Section 5.2.1). SLEMI then applies standard Simulink constructs (Model Reference) to reference the new model from the original model [93]. While Simulink does not propagate datatypes and other block attributes across such model-reference boundaries, SLEMI again leverages its datatype inference and annotation database from preprocessing.

Second, SLEMI's *live fork* mutation forks an existing live connection, to feed the same data to a new signal path. The new path is live as it terminates in a new sink. However, this path is generated to be not observable, as the sink is an assertion block that is designed to be always true. Concretely, before the new assertion block SLEMI currently adds a sequence of Simulink Math Operations blocks. These additions are EMI and have no effect on the produced traces.

Finally, SLEMI's *live-path mutation* currently focuses on the special case of replacing a top-level zombie block within a live path with another block that is expected to constantly produce the top-level zombie's default value. For example, in the Fig. 5.4a example model[4],

[4]For brevity Fig. 5.4 and subsequent figures omit the TypeCast blocks added by preprocessing.

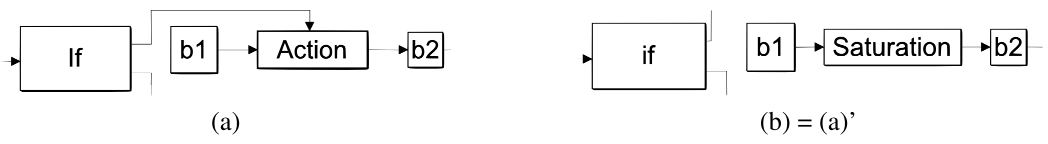

Figure 5.4: Example live-path mutation (b) that replaces a top-level zombie block (Action) in the $b1$ to $b2$ live path with a live Saturation block that mimics the default behavior of the replaced top-level zombie block.

SLEMI replaces the top-level zombie *Action* block with a live Saturation block, yielding Fig. 5.4b. The Saturation block constantly feeds the replaced block's default value to its live successor block $b2$.

5.3.4 Keeping Mutations EMI-preserving

Beyond designing individual mutation operations to preserve EMI, SLEMI applies additional rules across mutations. Besides mutations maintaining standard type rules, the following focuses on rules specific to CPS models.

Avoid Algebraic Loops

A SLEMI mutation should not introduce an *algebraic loop*, i.e., a circular data dependence path on which all blocks are *direct feed-through* [53]. On such a loop Simulink would need a block b's output value to compute b's input value. While Simulink can solve some algebraic loops, doing so is computationally expensive, so SLEMI avoids algebraic loops.

Specifically, SLEMI avoids replacing blocks that are not direct feed-through with direct feed-through blocks, to not turn a benign data dependence loop into an algebraic loop. For example, the Fig. 5.5a model contains a benign loop between Add and Delay, where the latter delays returning its input as an output to the next sample time. Since Delay is not direct feed-through, SLEMI will not replace it with a direct feed-through block such as another Add.

Avoid Invalid Execution Order Priority

During compilation, CPS tool chains determine the order in which to compute block outputs according to the model's data-dependencies, optional user-requested block execution

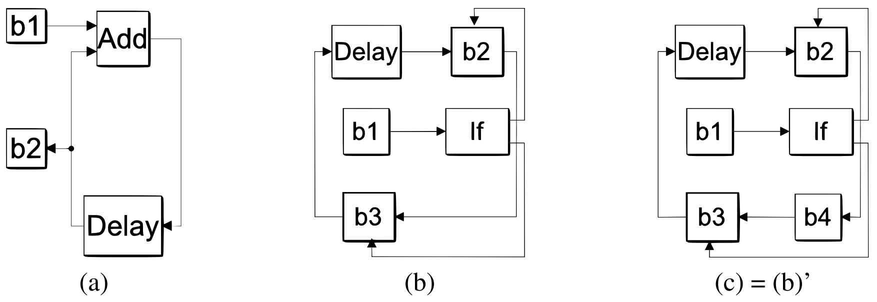

Figure 5.5: Example benign data dependence loop (a). Adding b_4 to (b) yields an invalid execution order priority in (c).

priorities, and language semantics. For example, Simulink orders all block output computations within one block priority level before the blocks of the next level. A Simulink rule we found out via initial feedback on a bug report is that an *If* block's action subsystems (branches) currently receive the same exclusive priority.

This rule disallows placing a block in a path between *If* block's action subsystems. In the valid Fig. 5.5b example model, the $b2$ and $b3$ action subsystems receive the same priority. Simulink thus first computes $b2$ then $b3$. But adding the $b4$ block outside these subsystems in Fig. 5.5c yields a compile error, as Simulink assigns $b4$ a different priority. So the $b4$ computation would happen either before or after both of $b2$ and $b3$, while the data dependencies call for $b4$ being computed between $b2$ and $b3$.

5.4 Evaluation

To evaluate our EMI-based mutation strategies in terms of their runtime and bug-finding capabilities, we explore the following research questions.

RQ1 How does SLEMI's runtime compare to SLforge?

RQ2 Can SLEMI find new CPS tool chain bugs?

RQ3 Does SLEMI find bugs that SLforge misses?

5.4.1 Seed Models and Their Input Values

The first source of seed models is a large corpus of some 1k user-created, publicly available Simulink models [18]. Of these 1k models, given our toolbox licenses, we could run 545 models with our Simulink installation. 18 of these models are interactive (i.e., they halt to wait for user input through a terminal or GUI) and thus we discarded them. If a (non-interactive) corpus model accepts inputs we used default 0 values.

Other corpus models have a long simulation duration (including *infinity*), which is also not desirable, so we limited their simulation duration to 10 seconds (since most of the models have this default duration [18]). Only 16 of the corpus models we were able to run had top-level zombie or nested zombie blocks.

SLforge-generated models had more top-level zombie and nested zombie blocks. SLforge generates model plus corresponding inputs, which can be readily used together. In a sample of some 150 models that is representative of the SLforge-generated models we used in our experiments, each model (except three) had at least one top-level zombie or nested zombie block (median value: 26.4% of the blocks in a model are such blocks). Besides enabling a variety of SLEMI mutations, SLforge-generated models are attractive for differential testing, as they have deterministic outputs by construction.

5.4.2 Evaluation Setup

The SLEMI prototype tool for finding bugs in Simulink is implemented in MATLAB on top of the *Parallel Computing Toolbox*. While in production mode the model mutation (and caching) jobs run in parallel, for debugging one can also configure SLEMI to mutate models sequentially in an *interactive* mode—pausing after desired mutation operations and highlighting the changes. SLEMI and all experimentation data are open source and freely available at GitHub (currently anonymously to support double-blind paper review) [5].

For our experiments we used the latest SLforge version (in its default configuration) and MATLAB releases R2017a, R2018a, and R2018b [15]. To evaluate SLforge and SLEMI side-by-side, we ran them separately in two otherwise idle machines (each with four Intel i74790 CPUs at 3.60 GHz, 64-bit Ubuntu 16.04, and 12 GB RAM).

To isolate EMI's impact from differential testing, for SLEMI we only compared mutant with seed on a single configuration (Normal mode with Optimization off). In contrast, since SLforge emphasizes differential testing, for SLforge we used all four of its differential testing configurations on each model (Normal + Optimization off, Accelerator + Optimization off, Normal + Optimization on, Accelerator + Optimization on).

5.4.3 Mutating is Faster than Generating (RQ1)

To explore SLEMI's runtime characteristics, we measured both how SLEMI's runtime scales with model size (measured as number of model blocks [67]) and how long each SLEMI phase takes. For this experiment, we used our 150 valid representative generated seed models of various sizes (from 100 to some 3k blocks, average 989). Based on earlier work these 150 models are similar to the non-toy models in the largest public corpus of open source Simulink models [17].

From these seed models we then generated 500 mutants by sampling uniformly from the seeds, creating some 3.5 mutants per seed on average. When reporting mutant creation results, we report the mean of all mutants generated for a seed.

During initial experiments we realized that individual Simulink tool chain phases may produce conflicting results. For example, a model that failed Simulink compilation when collecting block attributes successfully compiled for simulation. One such issue led to a confirmed bug report (Case 03213776). To catch such bugs the SLEMI implementation performs several tasks separately that could conceptually be combined into one phase, such as compilation to infer datatypes and execution for coverage collection.

Fig. 5.6 shows the number of mutation operations per model which is a (user configurable) fraction of the number of model blocks available for mutation (less blocks not mutated to keep the mutant EMI-preserving) along with SLEMI phases' runtime. The average phase runtimes (in seconds) were running the seed (51.7), collecting coverage (93.2), addressing datatype and sample time inference (94.5), generating (on average) 3.5 mutants (19.7), average mutant runtime (26.4), and average differential testing the seed with one of its mutants (169.7).

Overall, differential testing was the longest-running phase (average 41% of total run-

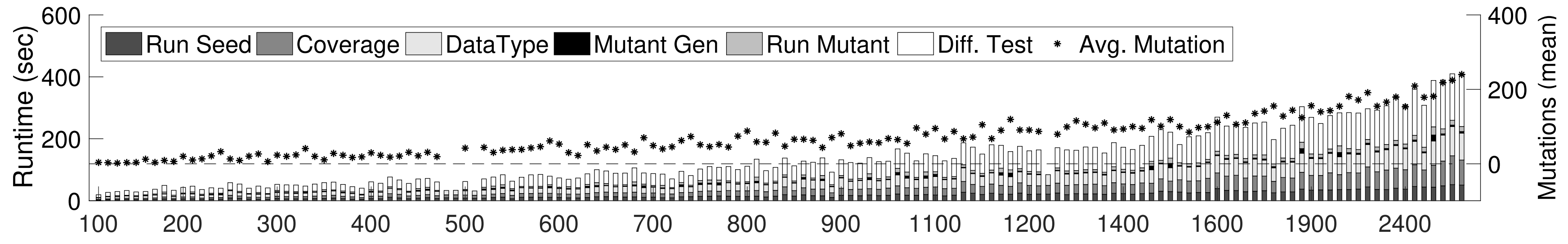

Figure 5.6: Cumulative runtime of SLEMI's three seed preprocessing and three per-mutant phases; data point = single seed and its mutants; * = average mutation operations that led to a seed's mutants; X-axis = blocks per seed model; data points/bars ordered by x-value and into correct 100-block seed bin, but beyond that not shown on precise X-axis location to improve readability; stacked bar shows SLEMI phases from running seed (bottom) to differential testing (top).

time) in each experiment, while mutant generation on average consumed only 2.4%. While the last three phases (mutant generation to differential testing) run once per mutant, all mutants of a seed share the first three phases (up to inferring the seed's datatypes and sample times).

In contrast, the state-of-the-art random Simulink model generator SLforge on average took about 470 seconds just to generate a single valid model. For this experiment, SLforge generated 167 models that were very similar to the models used as seeds in the SLEMI experiment. In other words, mutating an existing model in SLEMI was much faster than generating a fresh model with SLforge.

5.4.4 SLEMI Found New Bugs in Simulink (RQ2)

To date we have reported 14 unique issues to MathWorks Support, who confirmed 10 as a bug, of which 3 were already known to MathWorks (we re-discovered them independently)[5]. Two are still under investigation. Table 5.1 summarizes our reports. Following are details of some reports.

Case 03580171: Live-hierarchy extraction → Compile Error (New Bug) After mutating a seed by moving one of its blocks to a separate model file and then referencing the new model from the seed, Simulink inherited a different sample time for the block, resulting in an error. However, this is not expected since the block supports sample time inheritance for model referencing per documentation.

Case 03568445: Live Mutation → Block Output Mismatch (Known Bug) Having independently re-discovered a known issue in the Simulink Unary Minus block (where its output diverges for very small floating point inputs between Normal and Accelerator mode), we implemented mutation by adding such math operation blocks in live signal paths followed by assertion logic, to validate the added blocks' characteristics. This EMI-based mutation also reproduced the bug without entailing differential testing varying tool chain configurations (i.e., simulation modes). R2019a fixes this bug.

[5]MathWorks has fixed one of the bugs (TSC 03568445) in a future release.

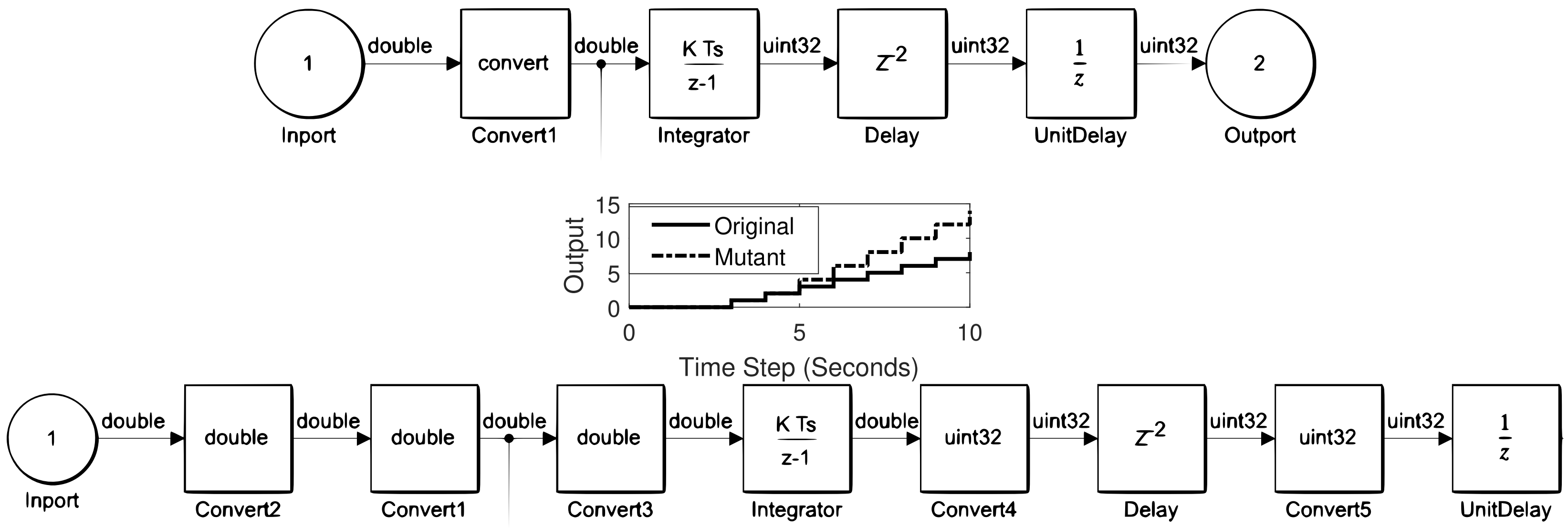

Figure 5.7: #03404633 (condensed): In a live region, Integrator's (topmost figure) uint32 output differs from the explicitly converted (via Convert4, figure at the bottom) output, as shown in the middle figure. The mutant has Data Type Conversions added after every block in the seed. Integrator in both versions received same input, hence selecting different types for internal calculation is the likely root cause.

Table 5.1: SLEMI-discovered issues: *TSC* = *Technical Support Case* number from MathWorks; *MW* = feedback from MathWorks on bug report (*N* = new confirmed bug, *K* = known bug, *F* = false positive, *?* = under investigation); *C* = bug reported when compiling mutant; *R* = bug reported at mutant runtime; *EMI* = EMI independently discovers, *T* = differential testing independently discovers, *S* = missing or hard to find specification (e.g., if only specified in a block-configuration GUI wizard). All bugs exist in R2018a except 03210493 (R2017a).

TSC	Summary	MW	Kind
03205823	Incorrect type inference after replacing block with type-compatible Ground block	N	EMI, T, C
03210493	After mutation Simulink does not eliminate dead Action Subsystem	K	EMI, C
03213776	Valid model stops compiling when collecting inferred properties	N	EMI, T, C
03259942	Invalid data-dependency loop for Action subsystem after mutation; undocumented specification	?	EMI, S, C
03404633	Wrong output for a Discrete Integrator block due to using different datatype at the mutant	?	EMI, R
03416784	Output discrepancy for Sin block after zero-crossing detection	F	EMI, R
03475044	Datatype inconsistency after enabling signal logging, due to complex type inference heuristics	N	EMI, T, C
03486057	When annotating sample time *getSampleTime* API does not return correct sample time	N	EMI, C
03486114	During type annotation Discrete Transfer Function does not accept a valid type. Only way to specify type is through controlling its input signal. The rule is specified in a GUI wizard.	F	EMI, S, C
03489578	Block does not respect sample time inference diagnostic command *InheritedTsInSrcMsg*	K	EMI, C
03489586	Signal Editor block errors-out due to unrelated configuration when specifying sample time	N	EMI, C
03568445	Behavior difference for Unary Minus operator when feeding small floating-point number	K	EMI, T, R
03580171	Incorrect sample time inference after live mutation by moving blocks via model-referencing	N	EMI, C
03785381	Incorrect block output range during live mutation	N	EMI, C

Case 03404633: Adding Data Type Conversion → Block Output Mismatch (Pending)

In the (condensed) seed in Fig. 5.7, Simulink infers an uint32 type for the discrete integrator block Integrator since its successor expects this input type. Inserting Data Type Conversion Convert4 between these two blocks (during pre-processing) yields the mutant.

However, these two models differ in Integrator's output values. Fig. 5.7 shows how they start diverging at simulation time step 5.

We think these values should not diverge. When manually minimizing the seed and mutant model, the two minimized models only differ in who is performing the type conversion, the discrete integrator itself (seed) or an added explicit type converter (mutant). We suspect that since in the mutant Integrator used a double type for both its input and output, it preserved its output in "full precision" whereas in the original model it lost precision due to converting from double (input) to uint32 (output), which could have been prevented by storing the data temporarily using double types and only converting to uint32 when emitting output.

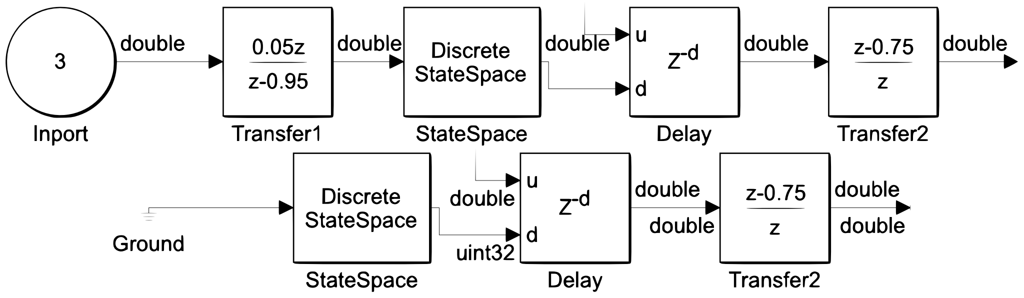

Figure 5.8: Case #03205823 (condensed) removes dead blocks Inport and Transfer1 in the seed model (top) and replaces them with the type-compatible (double) Ground (bottom) where Simulink failed to infer correct datatypes for all of the blocks, i.e., it inferred uint32 at Delay's input and propagated it back to the out-port of StateSpace which does not accept it.

Case 03205823: Nested Zombie Mutation → Compile Error (Likely New Bug) Fig. 5.8 shows condensed versions of the seed model (left) and the mutant model (right). This mutation replaced the nested zombie blocks Inport and Transfer1 with a Ground block that supports the output datatype double. However after adding it Simulink back-propagated uint32 to its predecessor, a DiscreteStateSpace block that does not accept this type, consequently yielding a compile error. MathWorks considers addressing this issue in a future release.

Case 03210493: Disconnecting block → Compile Error (Known Bug) In this mutation we disconnected a nested zombie *Action Subsystem* from its predecessors, successors, and its driving *If* block. Simulink did not remove the block, resulting in a compilation error in version R2017a. This error was unexpected since (1) no If block was connected to the subsystem meaning the block would never get executed and (2) the block was also dead so the Dead Block Reduction optimizer should have eliminated the subsystem.

Upon further investigation, MathWorks Support identified not explicitly specifying the block datatypes of the blocks in the Action subsystem as the root cause and suggested explicitly specifying the types as a workaround for the datatype inference limitations. Accordingly, to minimize datatype inference we now preprocess the models and annotate datatypes for all of the blocks in a seed (Section 5.3.1). We independently re-discovered this bug. Simulink R2018a fixed it.

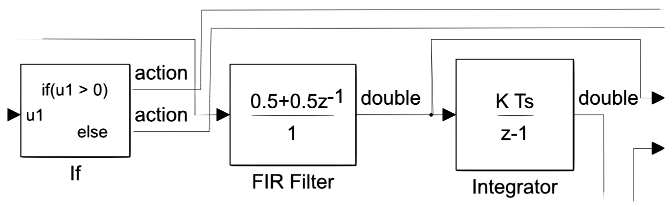

Figure 5.9: Case #03213776 (condensed): Simulink infers different output types for FIR FIlter across different tool chain settings: double in Normal mode vs. uint32 when compiling to collect the inferred block properties.

Case 03213776: Nested Zombie Mutation → Mismatch in Different Tool Chain Configurations (Likely New Bug) The excerpted model in Fig. 5.9 compiled and ran without errors using Simulink's *Normal* mode. But it produced a compilation error when we attempted collecting block properties after nested zombie block mutation, which is unexpected as models that simulate without errors should not raise errors when compiling for collecting these properties. MathWorks Support confirmed that these two different workflows use two different heuristics for datatype propagation and would consider making the results consistent in future releases.

5.4.5 SLEMI Finds Bugs Missed by SLforge (RQ3)

To compare the EMI-based mutation in SLEMI to plain differential testing in SLforge on bug finding efficiency, we also ran SLforge with similar resources. Specifically, while our

SLEMI experiments used under 200 CPU hours, we gave SLforge's default configuration over 300 CPU hours to find bugs. Of these, SLforge spent 80% on model generation and 20% on differential testing.

Compared to SLEMI's 10 unique bugs, in this experiment SLforge found two unique new bugs, which are a subset of the bugs SLEMI found with fewer resources. In addition to the two bugs SLforge found, upon manual inspection we observed that SLforge could have hypothetically found an additional two bugs we initially identified via EMI.

5.4.6 Threats to Validity

Both SLforge and SLEMI are prototype tools that only support a subset of the Simulink language and libraries. From a bug-finding perspective it is encouraging that these tools were still able to find several confirmed bugs in a widely used (and tested) commercial CPS tool chain.

A key threat to the validity of our tool comparisons is that the results just represent a particular implementation of the underlying bug-finding techniques for one particular CPS language, evaluated with models produced by a single seed model generator. Different implementation and evaluation choices could influence the tools' bug-finding abilities significantly. So without more tool implementations and experiments it would be premature to rule one tool chain testing technique superior to the other. In any case, the common best practice in software validation applies also to finding CPS tool chain bugs, i.e., to use all available bug-finding tools.

Since complete specifications for commercial CPS tools are not publicly available and our experiments only involved SLforge-generated models that cover a subset of the Simulink functionalities, the EMI approaches may not generalize to other CPS models. Although our approach has already found bugs covering widely-used Simulink libraries(from [17]), we consider experimenting with other libraries, user-created models and other CPS tools future work.

5.5 Related Work

Following is related work on mutating CPS models, EMI-based mutation for finding bugs in compilers for procedural languages (i.e., C and OpenCL), finding CPS development tool bugs without using model mutation, and finding bugs in CPS models.

Mutating CPS Models As discussed throughout the paper, the most closely related work is SLforge. While it mostly focuses on how to create random valid Simulink models for differential testing, SLforge also contains a very restricted version of Simulink model mutation. Specifically, for a given seed model, SLforge performs a single mutation operation, which (statically) deletes all dead blocks. In contrast, SLEMI takes into account model profiling data and performs several novel EMI-based mutation techniques that address CPS modeling challenges not found in procedural code, including zombie regions and sample time inference. Overall SLEMI's was more effective and efficient than SLforge.

Besides SLforge we are not aware of other work performing EMI-based mutation in CPS. However, the more restricted class of static equivalence-maintaining mutation has been of interest for several CPS-related tasks. For example, partial evaluation tries to minimize model size or simulation runtime while maintaining the model's execution behavior [66].

While partial evaluation produces a class of EMI-mutants, we consider them less promising for finding bugs in CPS tool chains, since modern tool chains likely already perform some forms of partial evaluation and thus resulting bugs are likely already known to the tool chain developers. A concrete example is MathWorks's Simulink Design Verifier [57], which, among others, has an option to detect and remove "dead logic", aka static nested zombie blocks.

Static equivalence-maintaining mutation is further interesting for model refactoring. For example, Tran et al. present an approach for composing elementary Simulink mutation operators into larger refactorings [94]. Users have to take care to ensure that a composed refactoring preserves model behavior. Also available are more specialized tools that are designed to preserve model behavior while improving a model's layout. For example, the

Auto Layout Tool can flatten a hierarchical model, by "inlining" a child model directly into its parent [68]. Other recent work transforms Simulink models [28] with the goal that the mutant approximates the seed model's behavior. These mutations are often more restrictive as they are done statically and they have not yet been applied for differential testing or finding tool bugs.

Model clones can have the same (or different) behavior as their seed. A recent taxonomy of Simulink model mutations for evaluating clone detection techniques was found to capture the manual edits performed on three Simulink projects [80].

Mutation testing aims at introducing small semantic changes to check if an existing test suite can detect the mutant's different execution behavior. In some sense mutation testing is the inverse of EMI-based mutation. For example, Zhan and Clark trace all paths from a change to outputs, to ensure that a change can be observed [100]. To select mutants efficiently, He et al. define an equivalence relation on models [40]. This equivalence notion is much coarser than ours, as it will consider equivalent two mutants whose execution behaviors differ widely, as long as both mutants are killed (detected) by the same test case.

EMI-based Mutation for Finding Bugs in Compilers for Procedural Languages To complement existing schemes for differential compiler testing, recent work has developed EMI-based mutation for C programs [48, 49, 85, 86] and OpenCL programs [51]. Overall these approaches have found in production-level compilers hundreds of previously unknown bugs, many of which the compiler developers have already fixed [83]. While early EMI-based mutation work focused on mutating dynamically dead program regions through both dead element removal [48] and addition [49], recent work also mutates live program paths [85].

These previous approaches have in common that their mutations target procedural languages (C and OpenCL) that have a complete specification. In contrast, this paper targets a flexible block diagram language that is widely used in CPS development, which (a) does not have a good specification and (b) has several key features not found in C or OpenCL, such as explicit notions of time, datatype inference, and zombie code.

Finding CPS Development Tool Bugs Without Model Mutation Earlier work has explored several avenues for finding bugs in CPS development tools. Most closely related is random model generation with "plain" differential testing (without EMI-based mutation), as implemented in CyFuzz [16] and SLforge [17]. Closely related to SLforge is a random model and differential testing tool by Nguyen et al. [65]. The tool first generates random hybrid automaton models and then HyST [6] translates the automaton models to a variety of CPS modeling languages including Simulink.

Other testing [82, 81, 76, 32] and analysis [32] schemes target select parts of a CPS development tool. For example, Stürmer et al. test optimization rules of code generators utilizing graph grammars [82, 81]. Fehér et al. model the data-type inferencing logic of Simulink blocks [32].

Analyzing and Finding Bugs in CPS Models Finally, while this paper looks for compiler bugs in CPS tools, a complimentary line of work analyzes and looks for bugs in CPS models [52, 46, 4, 34, 78, 3, 102, 10, 57]. For example, MathWorks's Simulink Design Verifier [57] uses static analysis to identify design errors in Simulink models, such as array access violations, division by zero static, integer overflow, and static nested zombie blocks. Similarly, DSVerifier [10] applies symbolic model checking based on SAT and SMT solvers to find design errors in digital systems.

Related work generates or evaluates the quality of test cases for CPS models [30, 9, 58, 79, 33], e.g., via mutation testing of Simulink models [9]. Other related work synthesizes controllers that are correct by design [74, 1]. While these directions are important, they are distinct from our work, which focuses on finding compiler bugs in CPS development tools rather than analyzing and testing the CPS models.

5.6 Conclusions

Finding bugs in commercial cyber-physical system development tools such as MathWorks Simulink is important in practice, as these tools are widely used to generate embedded code that gets deployed in safety-critical applications such as cars and planes. Equivalence Modulo Input (EMI) based mutation is a new twist on differential testing that promises lower use of computational resources and has already been successful at finding bugs in compilers for procedural languages. To provide EMI-based mutation for differential testing of cyber-physical system development tools, this paper has developed several novel mutation techniques. These techniques deal with CPS language features that are not found in procedural languages, such as an explicit notion of execution time and zombie code, which combines properties of live and dead procedural code. In our experiments the most closely related work SLforge found two bugs in the Simulink tool. In comparison, SLEMI found a super-set of issues, including 9 confirmed as bugs by MathWorks Support.

Future work includes adopting SLEMI to the closely related CPS modelling languages, including Simulink Stateflow [39] and the Simulink/Stateflow subset TargetLink [31].

CHAPTER 6

CONCLUSIONS

In this dissertation we explored novel techniques to automatically test a widely used commercial cyber-physical system (CPS) development tool, Simulink. Using such CPS development tools engineers and researchers design CPS using graphical block diagram models, simulate and even generate code and executables which are often deployed in safety-critical embedded hardware including cars and planes. Since CPS tool chain bugs may propagate to the automatically generated artifacts, it is crucial to eliminate bugs from development tools like Simulink. While formal verification of an entire CPS development tool would be ideal, unfortunately, such verification is inhibited by the scale of the development tool and also by the unavailability of complete and updated formal specifications of the CPS development tool. Randomized differential testing, on the other hand, does not require complete, updated specifications of the compiler system under test and has been effective in finding previously unknown bugs in major production compilers like GCC and LLVM, which motivated us to explore these techniques in the commercial CPS tool chain testing domain.

Verification and validation efforts in the CPS domain have targeted the CPS models whereas in this dissertation we are keen to develop techniques to find compiler bugs in commercial CPS development tools like Simulink. Existing compiler testing through random program generation works have targeted textual programming language (e.g. C and Java) compilers. Since graphical CPS modeling language like Simulink differs significantly than textual languages like C, we first examine the feasibility of applying randomized differential testing for CPS development tools and present CyFuzz, the first known differential testing framework to test Simulink. Although CyFuzz addressed many of the unique challenges in randomized generation of valid Simulink programs, it was not powerful enough to generate rich Simulink models to opportunistically find new compiler bugs.

To evaluate the effectiveness of a random Simulink model generator and even use insights to guide such a generator to create models similar to those designed by engineers and researchers, we have crafted the largest corpus of publicly available Simulink models to date. While similar corpus building efforts exist for other programming languages (e.g. Java), to the best of our knowledge no such corpus existed for Simulink models. Using insights from this corpus we next present the state of the art random, valid Simulink model generator SLforge which addresses several key CyFuzz limitations.

While complete and updated formal specifications for commercial CPS development tools like Simulink are not available, a random model generator could still utilize the semi-formal Simulink specifications made available through web-based documentations. Instead of leveraging such specifications, however, CyFuzz utilized heuristics to iterate over invalid Simulink models hoping to fix modeling errors to eventually get these accepted by the Simulink compiler as valid models. While this relatively simple feedback-driven approach helped creating valid Simulink models, the generated models are not expressive enough to find new Simulink bugs. Leveraging the semi-formal specifications SLforge now not only generates models rich in language elements but has also found new compiler bugs in various Simulink releases, proving its bug-finding capability.

Lastly, we explored a recent variation of differential testing, namely equivalence modulo input (EMI)-based compiler testing. EMI-based testing has been effective in finding compiler bugs missed by plain differential testing alone in recent C and OpenCL compiler testing projects. EMI-based mutation of Simulink models requires addressing zombie code which has properties of both live and dead code. Furthermore, unlike existing works EMI-based mutation of Simulink models has to deal with an explicit notion of time and CPS development tool features like data-type inference which we address in this work for the very first time. Our state-of-the-art EMI-based Simulink model mutator is more efficient than SLforge as it consumes less computing resource to mutate a model compared to generate an entire random model using SLforge and is also more effective in finding new compiler bugs which SLforge cannot find. The tools developed in this research are open source and have collectively found 18 unique bugs to date in various production versions of Simulink, the widely used commercial CPS development tool.

Although Simulink is the most widely used CPS development tool, future research in the commercial CPS tool chain testing direction may target other CPS development tools or extend the coverage of Simulink features supported by our random model generator and mutator tools. Exploring more EMI-based mutation techniques which are effective in finding compiler bugs is another exciting and promising line of future work. Lastly, continuously extending the large-scale corpus of Simulink models presented in this work would benefit the entire CPS research and tool development community.

BIBLIOGRAPHY

[1] Alessandro Abate, Iury Bessa, Dario Cattaruzza, Lennon C. Chaves, Lucas C. Cordeiro, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. DSSynth: An automated digital controller synthesis tool for physical plants. In *Proc. 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 919–924, November 2017.

[2] H. Alemzadeh, R. K. Iyer, Z. Kalbarczyk, and J. Raman. Analysis of safety-critical computer failures in medical devices. *IEEE Security Privacy*, 11(4):14–26, July 2013.

[3] Rajeev Alur. Formal verification of hybrid systems. In *Proc. 11th International Conference on Embedded Software, (EMSOFT) 2011*, pages 273–278. ACM, October 2011.

[4] Rajeev Alur, Aditya Kanade, S. Ramesh, and K. C. Shashidhar. Symbolic analysis for improving simulation coverage of Simulink/Stateflow models. In *Proc. 8th ACM & IEEE International Conference on Embedded Software (EMSOFT)*, pages 89–98. ACM, October 2008.

[5] Anonymous. Emi-based validation of commercial cps tool chains. https://github.com/cyemi/slsf_randgen/wiki, 2019. Accessed Jan. 2019.

[6] Stanley Bak, Sergiy Bogomolov, and Taylor T. Johnson. Hyst: A source transformation and translation tool for hybrid automaton models. In *Proc. 18th International Conference on Hybrid Systems: Computation and Control (HSCC)*, pages 128–133. ACM, 2015.

[7] Boris Beizer. *Software testing techniques*. Van Nostrand Reinhold, second edition, June 1990.

[8] Olivier Bouissou and Alexandre Chapoutot. An operational semantics for Simulink's simulation engine. In *Proc. 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES)*, pages 129–138. ACM, June 2012.

[9] Angelo Brillout, Nannan He, Michele Mazzucchi, Daniel Kroening, Mitra Purandare, Philipp Rümmer, and Georg Weissenbacher. Mutation-based test case generation for Simulink models. In Frank S. de Boer, Marcello M. Bonsangue, Stefan Hallerstede, and Michael Leuschel, editors, *Formal Methods for Components and Objects: 8th International Symposium, FMCO 2009, Eindhoven, The Netherlands, November 4-6, 2009. Revised Selected Papers*, pages 208–227. Springer, 2010.

[10] Lennon C. Chaves, Iury Bessa, Lucas C. Cordeiro, Daniel Kroening, and Eddie Batista de Lima Filho. Verifying digital systems with MATLAB. In *Proc. 26th*

ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pages 388–391, July 2017.

[11] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. An empirical comparison of compiler testing techniques. In *Proc. 38th International Conference on Software Engineering (ICSE)*, pages 180–190. ACM, 2016.

[12] R. J. Chevance and T. Heidet. Static profile and dynamic behavior of Cobol programs. *SIGPLAN Not.*, 13(4):44–57, April 1978.

[13] Shafiul Azam Chowdhury. Automatically finding bugs in commercial cyber-physical system development tool chains. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*, ICSE '18, pages 506–508. ACM, 2018.

[14] Shafiul Azam Chowdhury. Understanding and improving cyber-physical system models and development tools. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*, ICSE '18, pages 452–453. ACM, 2018.

[15] Shafiul Azam Chowdhury et al. Slforge web site. https://github.com/verivital/slsf_randgen/wiki. Accessed Jan. 2019.

[16] Shafiul Azam Chowdhury, Taylor T. Johnson, and Christoph 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, October 2016.

[17] Shafiul Azam Chowdhury, Soumik Mohian, Sidharth Mehra, Siddhant Gawsane, Taylor T. Johnson, and Christoph 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)*, pages 981–992. ACM, May 2018.

[18] Shafiul Azam Chowdhury, Lina Sera Varghese, Soumik Mohian, Taylor T. Johnson, and Christoph 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)*, pages 45–48. ACM, May 2018.

[19] Christian S. Collberg, Ginger Myles, and Michael Stepp. An empirical study of Java bytecode programs. *Softw., Pract. Exper.*, 37(6):581–641, 2007.

[20] Mirko Conrad. Testing-based translation validation of generated code in the context of IEC 61508. *Formal Methods in System Design*, 35(3):389–401, December 2009.

[21] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction To Algorithms*. MIT electrical engineering and computer science series. MIT Press, 2001.

[22] Christoph Csallner and Yannis Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software—Practice & Experience*, 34(11):1025–1050, September 2004.

[23] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 95–105, New York, NY, USA, 2018. ACM.

[24] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. Testing static analyzers with randomly generated programs. In *Proc. 4th NASA Formal Methods Symposium (NFM)*, pages 120–125. Springer, April 2012.

[25] Yanja Dajsuren, Mark G.J. van den Brand, Alexander Serebrenik, and Serguei Roubtsov. Simulink models are also software: Modularity assessment. In *Proc. 9th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*, pages 99–106. ACM, 2013.

[26] Florian Deissenboeck, Benjamin Hummel, Elmar Juergens, Michael Pfaehler, and Bernhard Schaetz. Model clone detection in practice. In *Proc. 4th International Workshop on Software Clones (IWSC)*, pages 57–64. ACM, 2010.

[27] Florian Deissenboeck, Benjamin Hummel, Elmar Jürgens, Bernhard Schätz, Stefan Wagner, Jean-François Girard, and Stefan Teuchert. Clone detection in automotive model-based development. In *Proc. of the 30th International Conference on Software Engineering (ICSE)*, pages 603–612. ACM, 2008.

[28] Joachim Denil, Pieter J. Mosterman, and Hans Vangheluwe. Rule-based model transformation for, and in Simulink. In *Proc. Symposium on Theory of Modeling and Simulation (TMS)*, pages 314–321. ACM, April 2014.

[29] Kyle Dewey, Jared Roesch, and Ben Hardekopf. Fuzzing the Rust typechecker using CLP (T). In *Proc. 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 482–493. IEEE, 2015.

[30] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, July 2008.

[31] dSPACE Inc. Targetlink. https://www.dspace.com/en/inc/home/products/sw/pcgs/targetli.cfm, 2019. Accessed Jan. 2019.

[32] P. Fehér, T. Mészáros, L. Lengyel, and P. J. Mosterman. Data type propagation in Simulink models with graph transformation. In *Proc. 3rd Eastern European Regional Conference on the Engineering of Computer Based Systems*, pages 127–137. IEEE, August 2013.

[33] K. Ghani, J. A. Clark, and Y. Zhan. Comparing algorithms for search-based test data generation of MATLAB Simulink models. In *2009 IEEE Congress on Evolutionary Computation*, pages 2940–2947, May 2009.

[34] Antoine Girard, A. Agung Julius, and George J. Pappas. Approximate simulation relations for hybrid systems. *Discrete Event Dynamic Systems*, 18(2):163–179, 2008.

[35] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.

[36] David E. Goldberg. *Genetic algorithms in search, optimization and machine learning*. Addison-Wesley, first edition, 1989.

[37] Mark Grechanik, Collin McMillan, Luca DeFerrari, Marco Comi, Stefano Crespi, Denys Poshyvanyk, Chen Fu, Qing Xie, and Carlo Ghezzi. An empirical investigation into a large-scale Java open source code repository. In *Proc. ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 11:1–11:10. ACM, 2010.

[38] C. Guger, A. Schlogl, C. Neuper, D. Walterspacher, T. Strein, and G. Pfurtscheller. Rapid prototyping of an EEG-based brain-computer interface (BCI). *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, 9(1):49–58, March 2001.

[39] Grégoire Hamon and John Rushby. An operational semantics for Stateflow. *International Journal on Software Tools for Technology Transfer*, 9(5):447–456, 2007.

[40] Nannan He, Philipp Rümmer, and Daniel Kroening. Test-case generation for embedded Simulink via formal concept analysis. In *Proc. 48th Design Automation Conference (DAC)*, pages 224–229. ACM, June 2011.

[41] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Proc. 21th USENIX Security Symposium*, pages 445–458. USENIX Association, August 2012.

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

[43] Xiaoqing Jin, Jyotirmoy V. Deshmukh, James Kapinski, Koichi Ueda, and Ken Butts. Benchmarks for model transformations and conformance checking. https://cps-vo.org/node/12108, 2017. Accessed Jan. 2019.

[44] Taylor T. Johnson, Stanley Bak, and Steven Drager. Cyber-physical specification mismatch identification with dynamic analysis. In *Proc. ACM/IEEE Sixth International Conference on Cyber-Physical Systems (ICCPS)*, pages 208–217. ACM, April 2015.

[45] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, March 2004.

[46] Aditya Kanade, Rajeev Alur, Franjo Ivancic, S. Ramesh, Sriram Sankaranarayanan, and K. C. Shashidhar. Generating and analyzing symbolic traces of Simulink/Stateflow models. In *Proc. 21st International Conference on Computer Aided Verification (CAV)*, pages 430–445. Springer, June 2009.

[47] Donald E. Knuth. An empirical study of FORTRAN programs. *Softw., Pract. Exper.*, 1(2):105–133, 1971.

[48] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 216–226. ACM, June 2014.

[49] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 386–399, New York, NY, USA, 2015. ACM.

[50] Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems: A Cyber-physical Systems Approach*. http://LeeSeshia.org, first edition, 2011.

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

[52] 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*, Feb 2017.

[53] MathWorks Inc . Algebraic loop concepts — MATLAB & simulink. https://www.mathworks.com/help/simulink/ug/algebraic-loops.html, 2019. Accessed Jan. 2019.

[54] MathWorks Inc. . Choose a solver — MATLAB & simulink. https://www.mathworks.com/help/simulink/ug/types-of-solvers.html, 2019. Accessed Jan. 2019.

[55] MathWorks Inc.. Sample time — MATLAB & simulink. https://www.mathworks.com/help/simulink/sample-time.html, 2019. Accessed Jan. 2019.

[56] MathWorks Inc. Blocks for which sample time is not recommended — MATLAB & simulink. https://www.mathworks.com/help/simulink/ug/sampletimehiding.html, 2019. Accessed Jan. 2019.

[57] MathWorks Inc. Simulink design verifier — MATLAB & simulink. https://www.mathworks.com/products/sldesignverifier.html, 2019. Accessed Jan. 2019.

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

[59] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[60] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of Unix utilities. *Commun. ACM*, 33(12):32–44, December 1990.

[61] M. Mohaqeqi and M. R. Mousavi. Sound test-suites for cyber-physical systems. In *10th International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 42–48, July 2016.

[62] Pieter J. Mosterman, Justyna Zander, Grégoire Hamon, and Ben Denckla. Towards computational hybrid system semantics for time-based block diagrams. In *Proc. 3rd IFAC Conference on Analysis and Design of Hybrid Systems (ADHS)*, pages 376–385. Elsevier, September 2009.

[63] National Instruments. Labview system design software. http://www.ni.com/labview/, 2019.

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

[65] Luan Viet Nguyen, Christian Schilling, Sergiy Bogomolov, and Taylor T. Johnson. Runtime verification of model-based development environments. In *Proc. 15th International Conference on Runtime Verification (RV)*, September 2015.

[66] Bentley James Oakes. Optimizing Simulink models. Technical Report SOCS-TR-2014.5, McGill University, 2014.

[67] Marta Olszewska, Yanja Dajsuren, Harald Altinger, Alexander Serebrenik, Marina A. Waldén, and Mark G. J. van den Brand. Tailoring complexity metrics for Simulink models. In *Proccedings of the 10th European Conference on Software Architecture Workshops, November 28 - December 2, 2016*, page 5, 2016.

[68] Vera Pantelic, Steven Postma, Mark Lawford, Monika Jaskolka, Bennett Mackenzie, Alexandre Korobkine, Marc Bender, Jeff Ong, Gordon Marks, and Alan Wassyng. Software engineering practices and Simulink: Bridging the gap. *International Journal on Software Tools for Technology Transfer (STTT)*, 20(1):95–117, February 2017.

[69] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Complete and accurate clone detection in graph-based models. In *Proc. 31st IEEE International Conference on Software Engineering (ICSE)*, pages 276–286, May 2009.

[70] Pierre Giroux. Grid-connected pv array - file exchange - MATLAB central. http://www.mathworks.com/matlabcentral/fileexchange/34752-grid-connected-pv-array, August 2017.

[71] A. C. Rajeev, Prahladavaradan Sampath, K. C. Shashidhar, and S. Ramesh. CoGenTe: A tool for code generator testing. In *Proc. 25th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 349–350. ACM, September 2010.

[72] Akshay Rajhans, Srinath Avadhanula, Alongkrit Chutinan, Pieter J. Mosterman, and Fu Zhang. Graphical modeling of hybrid dynamics with simulink and stateflow. In *Proc. 21st International Conference on Hybrid Systems: Computation and Control (HSCC)*.

[73] Steven Rasmussen, Jason Mitchell, Chris Schulz, Corey Schumacher, and Phillip Chandler. A multiple UAV simulation for researchers. In *AIAA Modeling and Simulation Technologies Conference and Exhibit*, page 5684.

[74] Pritam Roy, Paulo Tabuada, and Rupak Majumdar. Pessoa 2.0: A controller synthesis tool for cyber-physical systems. In *Proc. 14th ACM International Conference on Hybrid Systems: Computation and Control (HSCC)*, pages 315–316, April 2011.

[75] Jesse Ruderman. Introducing jsfunfuzz. https://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/, 2007.

[76] Prahladavaradan Sampath, A. C. Rajeev, S. Ramesh, and K. C. Shashidhar. Testing model-processing tools for embedded systems. In *Proc. 13th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 203–214. IEEE, April 2007.

[77] Flash Sheridan. Practical testing of a C99 compiler using output comparison. *Software: Practice & Experience (SPE)*, 37(14):1475–1488, November 2007.

[78] S. Sims, R. Cleaveland, K. Butts, and S. Ranville. Automated validation of software models. In *Proc. 16th Annual International Conference on Automated Software Engineering (ASE)*, pages 91–96, Nov 2001.

[79] A. Sridhar, D. Srinivasulu, and D. P. Mohapatra. Model-based test-case generation for Simulink/Stateflow using dependency graph approach. In *Proc. 3rd IEEE International Advance Computing Conference (IACC)*, pages 1414–1419, February 2013.

[80] Matthew Stephan, Manar H. Alalfi, and James R. Cordy. Towards a taxonomy for Simulink model mutations. In *Proc. 7th IEEE International Conference on Software Testing, Verification and Validation (ICST) Workshops*, pages 206–215. IEEE, March 2014.

[81] Ingo Stürmer and Mirko Conrad. Test suite design for code generation tools. In *Proc. 18th IEEE International Conference on Automated Software Engineering (ASE)*, pages 286–290, October 2003.

[82] Ingo Stürmer, Mirko Conrad, Heiko Dörr, and Peter Pepper. Systematic testing of model-based code generators. *IEEE Transactions on Software Engineering (TSE)*, 33(9):622–634, September 2007.

[83] Zhendong Su and Chengnian Sun. EMI compiler validation project. http://web.cs.ucdavis.edu/~su/emi-project/, 2019. Accessed Jan. 2019.

[84] Giancarlo Succi, Witold Pedrycz, Snezana Djokic, Paolo Zuliani, and Barbara Russo. An empirical exploration of the distributions of the Chidamber and Kemerer object-oriented metrics suite. *Empirical Software Engineering*, 10(1):81–104, 2005.

[85] Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. *SIGPLAN Not.*, 51(10):849–863, October 2016.

[86] Qiuming Tao, Wei Wu, Chen Zhao, and Wuwei Shen. An automatic testing approach for compiler based on metamorphic testing technique. In *Proceedings of the 2010 Asia Pacific Software Engineering Conference*, APSEC '10, pages 270–279, Washington, DC, USA, 2010. IEEE Computer Society.

[87] E. 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 *2010 Asia Pacific Software Engineering Conference*, pages 336–345, Nov 2010.

[88] The MathWorks Inc. How many blocks are in that model? mathworks blog. https://blogs.mathworks.com/simulink/2009/08/11/how-many-blocks-are-in-that-model/. Accessed Jan. 2019.

[89] The MathWorks Inc. Nasa hl-20 lifting body airframe - MATLAB & Simulink. https://www.mathworks.com/help/aeroblks/nasa-hl-20-lifting-body-airframe.html, 2017. Accessed Jan. 2019.

[90] The MathWorks Inc. Products and services. http://www.mathworks.com/products/, 2018.

[91] The MathWorks Inc. About data types in Simulink. https://www.mathworks.com/help/simulink/ug/working-with-data-types.html, 2019. Accessed Jan. 2019.

[92] The MathWorks Inc. Simulink coverage. https://www.mathworks.com/help/slcoverage/, 2019. Accessed Jan. 2019.

[93] The MathWorks Inc. Simulink documentation. http://www.mathworks.com/help/simulink/, 2019. Accessed Jan. 2019.

[94] Quang Minh Tran, Benjamin Wilmes, and Christian Dziobek. Refactoring of Simulink diagrams via composition of transformation steps. In *Proc. 8th International Conference on Software Engineering Advances (ICSEA)*, pages 140–145. IARIA, October 2013.

[95] U.S. Consumer Product Safety Commission (CPSC). Recall 11-702: Fire alarm control panels recalled by fire-lite alarms due to alert failure. http://www.cpsc.gov/en/Recalls/2011/Fire-Alarm-Control-Panels-Recalled-by-Fire-Lite-Alarms-Due-to-Alert-Failure, October 2010.

[96] U.S. National Highway Traffic Safety Administration (NHTSA). Defect Information Report 14V-053. http://www-odi.nhtsa.dot.gov/acms/cs/jaxrs/download/doc/UCM450071/RCDNN-14V053-0945.pdf, February 2014.

[97] U.S. National Institute of Standards and Technology (NIST). The economic impacts of inadequate infrastructure for software testing: Planning report 02-3, May 2002.

[98] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 283–294. ACM, June 2011.

[99] Justyna Zander, Ina Schieferdecker, and Pieter J. Mosterman. *Model-based testing for embedded systems*. CRC Press, first edition, 2011.

[100] Yuan Zhan and John A. Clark. Search-based mutation testing for Simulink models. In *Proc. Genetic and Evolutionary Computation Conference (GECCO)*, pages 1061–1068. ACM, June 2005.

[101] Hongyu Zhang and Hee Beng Kuan Tan. An empirical study of class sizes for large Java systems. In *Proc. 14th Asia-Pacific Software Engineering Conference (APSEC)*, pages 230–237, December 2007.

[102] Liang Zou, Naijun Zhan, Shuling Wang, and Martin Fränzle. Formal verification of Simulink/stateflow diagrams. In Bernd Finkbeiner, Geguang Pu, and Lijun Zhang, editors, *Proc. 13th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 464–481. Springer, October 2015.