Demo: SLEMI: Finding Simulink Compiler Bugs through Equivalence Modulo Input (EMI)

Shafiul Azam Chowdhury
Computer Science and Engineering Department
University of Texas at Arlington
Arlington, Texas, USA

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

Taylor T. Johnson
EECS Department
Vanderbilt University
Nashville, Tennessee, USA

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

ABSTRACT

This demo presents usage and implementation details of SLEMI. SLEMI is the first tool to automatically find compiler bugs in the widely used cyber-physical system development tool Simulink via Equivalence Modulo Input (EMI). EMI is a recent twist on differential testing that promises more efficiency. SLEMI implements several novel mutation techniques that deal with CPS language features that are not found in procedural languages. This demo also introduces a new EMI-based mutation strategy that has already found a new confirmed bug in Simulink version R2018a. To increase SLEMI's efficiency further, this paper presents parallel generation of random, valid Simulink models. A video demo of SLEMI is available at https://www.youtube.com/watch?v=oliPgOLT6eY.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; *Compilers*; *Model-driven software engineering*; • **General and reference** → Reliability; Verification; Performance.

KEYWORDS

Cyber-physical systems, differential testing, equivalence modulo input, model mutation, Simulink

ACM Reference Format:
Shafiul Azam Chowdhury, Sohil Lal Shrestha, Taylor T. Johnson, and Christoph Csallner. 2020. Demo: SLEMI: Finding Simulink Compiler Bugs through Equivalence Modulo Input (EMI). In *42nd International Conference on Software Engineering Companion (ICSE '20 Companion), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3377812.3382147

1 INTRODUCTION

Designing complex cyber-physical systems (CPS) using graphical block diagrams (e.g., Simulink models [5]) is a standard engineering practice, which enables simulation and automated code generation of CPS. Since these automatically generated artifacts are commonly deployed in safety-critical embedded hardware (e.g., cars and planes), it is crucial to eliminate compiler bugs from commercial CPS development tools (e.g., Simulink).

Ideally, one should formally verify the absence of bugs in an entire CPS development tool chain, which typically includes compilers, simulators, and code generators. This unfortunately is still not feasible due to the sheer code-base size of a commercial CPS development tool. Moreover, complete, updated, and official formal specifications of commercial CPS tools are not available, which inhibits their formal verification [2].

Automated *differential testing*, on the other hand, has been effective in finding compiler bugs in Simulink and in other textual programming languages (e.g., C and Java). *SLforge*, the state-of-the art Simulink differential testing tool uses an automated Simulink model generator to continuously produce valid Simulink models [2]. For any such generated model if we observe output divergence when simulating in varying compiler configurations (e.g., enabling and disabling optimization), we have likely found a compiler bug.

Although SLforge has discovered several new compiler bugs in various Simulink releases, the approach is slow, since generating large Simulink models from scratch is computationally expensive. A recent variation of differential testing, namely Equivalence Modulo Input (EMI), instead introduces small changes to an existing valid Simulink model (i.e., *seed*) and produces a *mutant* such that it is valid and functionally equivalent to the seed w.r.t. some input I common to both [7].

EMI-based automated testing of compiler tool chains is faster than differential testing via random generation alone and hence increases the likelihood of finding bugs within some compute budget. For example, our open-source and only known tool for EMI-based automated testing of the Simulink tool chain, SLEMI, has found 10 bugs confirmed by MathWorks Support in various Simulink versions, whereas in the same experiment the closely related tool SLforge found only 2 bugs using similar computational resources [3].

This demo presents SLEMI, which implements novel EMI-based mutation techniques for the commercial CPS modeling language Simulink. SLEMI addresses key differences between procedural code and CPS modeling, which includes an explicit notion of simulation time and zombie code [3]. This demo also presents a new live mutation strategy in SLEMI, which has already found a new, confirmed bug in Simulink R2018a. We highlight SLEMI components

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

ACM ISBN 978-1-4503-7122-3/20/05.
https://doi.org/10.1145/3377812.3382147

and implementation details that led to increased throughput by parallel processing of Simulink models. Lastly, to match SLEMI's seed model consumption, we also present our improvement over the SLforge tool, namely SLFORGE++, which generates random, valid Simulink models in parallel.

2 BACKGROUND

This section contains necessary background information on key CPS modeling features, zombie code, and recent approaches for finding bugs in the Simulink tool via differential testing and EMI-based model mutation.

2.1 CPS Models & Development Tool Chains

While in-depth descriptions of CPS languages are available elsewhere [2, 3, 5, 6], following are the key concepts. In a CPS development tool (e.g., Simulink), a user designs a CPS as a model m that consists of blocks and their connections. 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 (directed) *connection* edges.

A model m typically acquires its inputs from sensors, whose values it samples at a user-defined frequency. To affect its environment, a model typically has a set of *output blocks* (aka *sinks*) such as Figure 1's *Out*1 and *Out*2 blocks, which can emit model output values to a display, another model, or a hardware actuator.

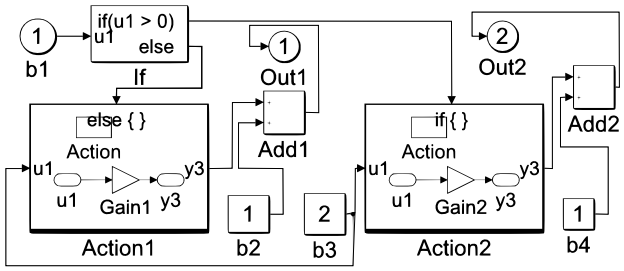

Figure 1: Example valid Simulink model: While *Action*1 is on a false-branch when $b1$ receives non-zero positive input, its values can still affect the outside world, making it a zombie.

Commercial CPS tools specify the *datatypes* each port of each block supports. If the user does not explicitly configure a port's datatype, then the CPS tool infers a concrete type (e.g., "double"). Besides flat models, CPS development tools offer hierarchical models, where the parent (e.g., *Action1* in Figure 1) to child model (e.g., *Gain1*) relation is acyclic.

Commercial CPS tool chain semantics are only defined via their code base [2]. We call a model *valid* if it can be compiled by a CPS tool chain without errors. Tools typically offer different simulation modes. For example, Simulink *Normal* mode "only" simulates blocks and *Accelerator* mode speeds up simulation by emitting code.

2.2 Zombies: Dead Code in Procedural vs. CPS

In block diagrams such as Simulink models, conditional execution differs significantly from procedural programming, which complicates the dead vs. live code distinction. For example, the (valid) Simulink model of Figure 1 simultaneously returns values from both its true and false if-then-else branches. In a procedural setting *Action*1 would be dynamically dead code and we could delete it for this execution trace. But in our block diagram setting *Action*1 is not dead. 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). Finally, a block is live if it has both a path to an output block (or another side-effect) and gets activated [3].

2.3 Testing Simulink with SLforge and SLEMI

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. In the CPS world, existing approaches for testing Simulink such as CyFuzz and SLforge have varied compiler optimization levels, simulation modes, and code generators [1, 2].

Equivalence modulo input (EMI)-based differential testing such as in SLEMI [3], on the other hand, uses a program m (aka seed) and one of its mutants n that is expected to be functionally equivalent to m on the given input I. Again, different results suggest a compiler bug.

3 SLEMI: TESTING SIMULINK VIA EMI

We consider two CPS models m and n (n obtained by mutating m) equivalent modulo a common sequence I of input vectors, if both models are valid, have the same output blocks, and the CPS tool chain semantics at all time steps prescribe equivalent values for all blocks b that are common to both models. Since Simulink supports floating-point datatypes, we compare block outputs via a configurable tolerance (10^{-16} by default) [3].

Figure 2 outlines our approach. SLEMI takes as input CPS models and their input values, which can come from a corpus of real-world models [4] or from a model generator such as SLforge. In a *preprocessing* phase, we first filter out invalid models and then execute each seed on its inputs to collect block-level coverage information, via *Simulink Coverage* to find zombie blocks. To achieve better runtime efficiency, SLEMI then performs several one-time *base mutations* and stores data in a persistent cache. We then mutate a model by removing and adding blocks.

3.1 Tools Overview: SLforge++ and SLEMI

Random program generators have significantly improved the bug-finding capability of differential testing [7]. SLEMI uses SLforge both to generate random Simulink models and to compare Simulink model execution traces. For this paper we have adapted SLforge to the *Stampede2* high-performance computing cluster of the Texas Advanced Computing Centre (TACC)[1]. The resulting SLFORGE++ tool can leverage TACC's many-core nodes to generate Simulink models in parallel and then add them to SLEMI's seed model corpus. Our tools run on Stampede2's *SKX* node (48 Intel Xeon Platinum 8160 cores; 192 GB RAM). Both SLEMI (6 kLOC) and SLforge++

[1] https://portal.tacc.utexas.edu/user-guides/stampede2

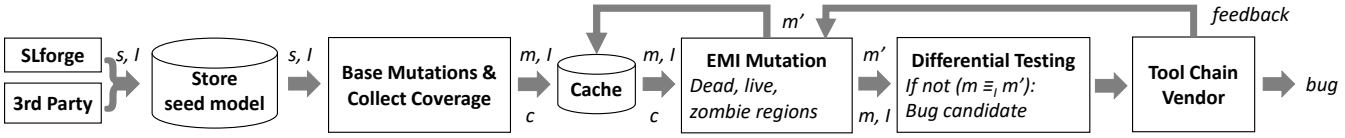

Figure 2: Overview: SLEMI first obtains seed model s with input vector I from a real-world corpus or a random generator (e.g., SLforge), performs one-time base mutations to yield model m, and collects m's coverage c (on I). An EMI-based mutation then yields a valid equivalent (on I) model m' (i.e., $m \equiv_I m'$) for finding tool chain bugs via differential testing. [3]

(9 kLOC) are implemented in MATLAB on top of the *Parallel Computing Toolbox*, and are freely available at GitHub[2].

SLEMI has three independent components: (1) First, an *Experimentation Framework* (3 kLOC) to support general-purpose experiments on Simulink models leveraging parallel computing. We utilize the framework for both analyzing Simulink models (e.g., to explore the distribution of zombie blocks in a corpus of Simulink models) and preprocessing seeds to speed-up overall runtime through several one-time base mutations.

(2) Second, our *EMI* component (2 kLOC) consumes the preprocessed seeds and outputs multiple mutants per seed by implementing various novel model mutation strategies. Again, we leverage parallel computing to both mutate multiple seeds and produce multiple mutants per seed. Each mutation strategy is implemented as an individual module such that users can configure which EMI strategies to perform during an experiment.

(3) Lastly, our *differential testing* component (1 kLOC) compares each of the generated mutants with its (preprocessed) seeds by simulating them in Simulink with an option to vary user-provided simulation modes and optimization settings. This is independent from the rest of the SLEMI components, i.e., it can also be triggered on an existing corpus of Simulink models, without generating mutants from them beforehand. SLEMI's differential testing improves upon SLforge's differential testing mechanism by (1) being modular and (2) supporting parallel processing of models, hence increasing throughput.

3.2 SLEMI's Experimentation Framework

During our implementation of SLEMI, we observed the absence of modular experimentation tools to facilitate analysis of Simulink models at scale. To address this issue, we have released a general-purpose experimentation framework as part of SLEMI, which analyzes and edits Simulink models in parallel.

3.2.1 Preprocessing Seed Models. Here, we further highlight our experimentation framework functionalities using preprocessing of seed models as example. SLEMI exposes most of the functionalities through a command line interface and configuration files. For example, users can configure a file system location, which SLEMI scans for Simulink models to use as seeds. Currently SLEMI supports local and the Lustre[3] distributed filesystem TACC uses. Users can filter out seed models by putting constraints on model modification date or its location in the filesystem.

Furthermore, using the framework users are able to design their own *modules* to analyze or mutate Simulink models and set dependencies between the modules. For example, we have developed the following modules and configured their dependencies: *AnalyzeSeeds* → *CollectCoverage* → *BaseMutation*. Meaning, for each of the seed models SLEMI first filters out invalid models through *AnalyzeSeeds*. *CollectCoverage* then profiles the valid seed models using *Simulink Coverage* and also collects datatype and sample-time information for each block in the model. Since *AnalyzSeeds* is a pre-requisite for *CollectCoverage*, the later module is only invoked when the first module terminates and can pass analysis results to the next module in its dependency configuration.

3.2.2 Efficiency through Caching. SLEMI's modular implementation of the experimentation framework and intermediate caching reduces development waste. For example, after introducing some changes (e.g., bug fixes) in the *BaseMutation* module, we only need to run this module since SLEMI caches output of the other modules it depends on (i.e., *CollectCoverage* and *AnalyzeSeeds*). This significantly reduces both development time (since running these dependencies would increase average runtime by about 150% [3]) and wasted computation by skipping resource-heavy modules.

SLEMI also deals efficiently with worker-node crashes, e.g., due to machine crashes and SLEMI development bugs. Specifically, in its parallel mode SLEMI processes each seed on its own "MATLAB worker", which may crash when running some experimentation framework module. When we invoke SLEMI next time, it automatically identifies those incomplete experiments and only re-runs the modules that had failed before completion.

3.3 Debugging and Analyzing Reports

While in production mode SLEMI processes seeds in parallel, in debug mode it processes models sequentially and can pause after every mutation operation, highlighting the affected blocks and then wait for user input to continue. For this visualization SLEMI uses Simulink's graphical user interface (GUI). Figure 3 shows an example.

After analyzing a seed model, SLEMI's experimentation framework summarizes the results (e.g., the zombie blocks available for mutation) using MATLAB's GUI. SLEMI also facilitates analyzing differential testing results interactively, which upon completion summarizes all of the likely bugs (i.e., whenever it has logged a mismatch between the block output of two models that should be EMI). Users then can choose a likely bug to inspect further by asking SLEMI to load the preprocessed seed and its mutants in a

[2]https://github.com/shafiul/slemi
[3]http://lustre.org/

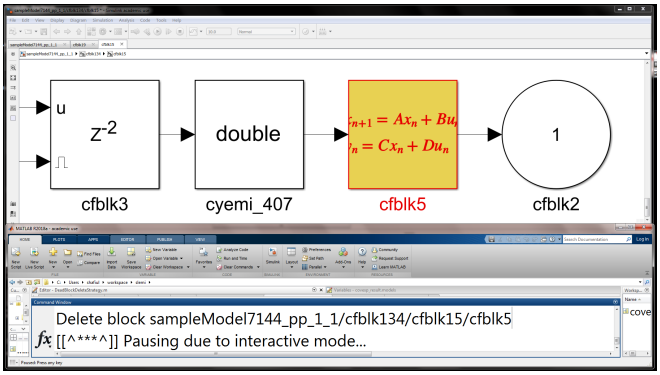

Figure 3: In interactive mode SLEMI pauses before each mutation operation and highlights the changes in a GUI: Here block `cfblk5` will be deleted upon user input.

GUI, along with all of the block outputs that diverge. SLEMI also integrates with Simulink's *Simulation Data Inspector* tool to visualize the block output discrepancies.

3.4 Finding Simulink Component Conflicts

Simulink compiles models for various reasons, e.g., to check model validity, collect inferred block attributes such as datatypes, simulate a model, and to collect coverage. We observed that these different compilations sometimes non-intuitively manifest deterministic ambiguous outcomes for the same seed model. For example, a model that fails to compile when collecting block attributes may execute successfully due to Simulink implementation differences in the compilation and execution phases. To better analyze these Simulink compile modes and find bugs in these tool chain components, for a seed model we individually invoke each of these phases (i.e., compiling, executing, and collecting coverage), which helped us discover a confirmed bug where compilation and execution outcomes were ambiguous.

3.5 Complex Live Mutations

In addition to SLEMI's existing mutation techniques [3], SLEMI now implements novel EMI-based mutation techniques that have already found a new confirmed bug. This mutation moves a block from some live model region into a subsystem that should always get activated, using Simulink *If* and *Action* constructs.

To synthesize a conditional expression for the *If* block that should always be evaluated to true, SLEMI dynamically collects some blocks' signal ranges, using Simulink's signal-range analysis. For example, $b_{min} <= b$ evaluates to true during an entire simulation for some block's output b where b_{min} is its minimum output during that simulation. This novel mutation strategy of moving a live model region into a new subsystem has already found one new bug confirmed by MathWorks Support, within Simulink's signal range analysis component.

Similarly, SLEMI synthesizes new zombie regions, places abort constructs within them, and finally places the newly constructed zombie region within existing live and zombie regions in the seed. This mutation is EMI since these new abort constructs should never get activated (unless due to a Simulink bug).

4 CONCLUSIONS

This demo presented usage and implementation details of SLEMI. SLEMI is the first tool to automatically find compiler bugs in the widely used cyber-physical system development tool Simulink via Equivalence Modulo Input (EMI). SLEMI implements several novel mutation techniques that deal with CPS language features that are not found in procedural languages. This demo also introduced a new EMI-based mutation strategy that has already found a new confirmed bug in Simulink version R2018a. To increase SLEMI's efficiency further, this paper presented parallel generation of random, valid Simulink models. All of our tools are open-source and freely available.

ACKNOWLEDGMENTS

The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper.

Christoph Csallner has a potential research conflict of interest due to a financial interest with Microsoft and The Trade Desk. A management plan has been created to preserve objectivity in research in accordance with University of Texas at Arlington policy.

The material presented in this paper is based on work supported by the National Science Foundation (NSF) under grant numbers 1527398, 1736323, 1910017, 1911017, and 1918450, the Air Force Office of Scientific Research (AFOSR) under contract number FA9550-18-1-0122, and a gift from MathWorks. The U.S. government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of AFOSR, NSF, or MathWorks.

REFERENCES

[1] Shafiul Azam Chowdhury, Taylor T. Johnson, and Christoph Csallner. 2016. 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, 46–60. https://doi.org/10.1007/978-3-319-51738-4_4

[2] Shafiul Azam Chowdhury, Soumik Mohian, Sidharth Mehra, Siddhant Gawsane, Taylor T. Johnson, and Christoph Csallner. 2018. Automatically finding bugs in a commercial cyber-physical system development tool chain with SLforge. In *Proc. 40th ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, 981–992.

[3] Shafiul Azam Chowdhury, Sohil Lal Shrestha, Taylor T. Johnson, and Christoph Csallner. 2020. SLEMI: Equivalence modulo input (EMI) based mutation of CPS models for finding compiler bugs in Simulink. In *Proc. 42nd ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM. To appear.

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

[5] MathWorks Inc. 2020. Simulink Documentation — MATLAB & Simulink. http://www.mathworks.com/help/simulink/. Accessed Feb 2020.

[6] Akshay Rajhans, Srinath Avadhanula, Alongkrit Chutinan, Pieter J. Mosterman, and Fu Zhang. 2018. Graphical modeling of hybrid dynamics with Simulink and Stateflow. In *Proc. 21st International Conference on Hybrid Systems: Computation and Control (HSCC)*. ACM, 247–252.

[7] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proc. ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 849–863.