SLEMI: Equivalence Modulo Input (EMI) Based Mutation of CPS Models for Finding Compiler Bugs in Simulink

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

Finding bugs in commercial cyber-physical system development tools (or "model-based design" tools) such as MathWorks's 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 (CPS) 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. In comparison, SLEMI found a super-set of issues, including 9 confirmed as bugs by MathWorks Support.

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. SLEMI: Equivalence Modulo Input (EMI) Based Mutation of CPS Models for Finding Compiler Bugs in Simulink. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380381

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

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380381

1 INTRODUCTION

Commercial cyber-physical system (CPS) development tools (or "model-based design" 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 [5, 17, 24, 57] and the tools' source code is not available either. While state-of-the-art CPS tool bug-finding approaches such as SLforge [13] 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, MathWorks's CPS development tool Simulink [32] is a de-facto industry standard in several safety-critical domains, including automotive, aerospace, and health care [63]. Engineers widely use Simulink to design, model, simulate, and generate embedded code from CPS models and deploy the generated code in safety-critical applications [6, 23, 48]. 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 [8].

To side-step the lack of CPS tool specifications, state-of-the-art CPS tool bug-finding approaches such as SLforge perform differential testing [43, 52, 62] on the CPS tool. By invoking two configurations of the same CPS tool that are expected to produce the same result 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 turns out to be 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 [28]. 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 [12, 13, 14].

While prior work has reported promising results on EMI-based mutation for finding bugs in C compilers [28, 29, 59], 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 model elements the compiler does not remove, and does not insert any model elements. 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 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 9 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 9 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] Available: https://github.com/shafiul/slemi

2 BACKGROUND

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

2.1 Block Diagrams and CPS Tool Chains

While in-depth descriptions of CPS languages are available elsewhere [13, 41, 44, 48, 49], the following are the key concepts. In a cyber-physical system (CPS) development tool (e.g., Simulink), a user designs a model m of the CPS as a diagram that consists of blocks and their connections. A block may accept 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 output port p_s, target block b_t, and its input port p_t [13].

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 [44]. To affect its environment, a model typically has a set of *output blocks* (or sinks) m_{out} such as Figure 1's $Out1$ and $Out2$ blocks, which can emit model output values to a display, another model, or a hardware actuator.

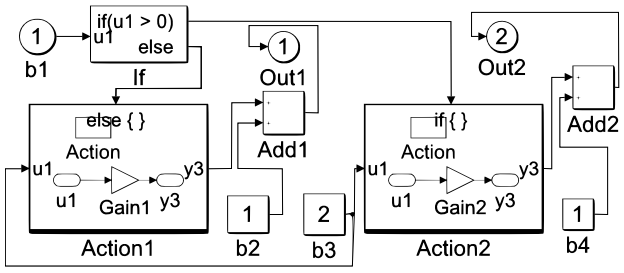

Figure 1: Example valid Simulink model: While $Action1$ is on a false-branch when $b1$ receives non-zero positive input, $Action1$'s values can still affect the outside world, making $Action1$ a zombie block.

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 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.

Tools such as Simulink are also known as model-based design (MBD) tools and have become popular in several domains, including automotive and aerospace engineering. A key reason for their wide use is that these tools allow non-software engineers to design complex software systems and compile them to low-level code.

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 new custom 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 publicly available specification that is complete, formal, and up to date. Besides partial informal descriptions, certain tool chain semantics are often defined in their code base [57]. 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 [20, 33].

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 Accelerator* mode produces a standalone executable for simulation[2].

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

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 [26]. 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 Figure 1 simultaneously returns values from both its true and false if-then-else branches. Specifically, 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 *Action*1 as the false-branch. Unlike in procedural programming, such a false-branch still has a user-configured default output value (for *Action*1 this default value is zero). The subsequent increment modeled by the *Add*1 block causes *Out*1 to constantly emit 1, which can affect the outside world.

[2]Simulink's embedded code generation worfklow for deployment on target platforms is distinct from these simulation modes and is outside the scope of this paper.

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. Instead, a block b is *dead* and can be removed only 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 *Action*1. 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., *Action*1).

*Action*1 is a *top-level zombie*, its (default) value can reach the outside world. In contrast, a *nested zombie* such as Figure 1's *Gain*1 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.

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., are expected to 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[3] 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 [12, 13, 15].

Equivalence modulo input (EMI)-based differential testing has typically 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

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

been recent interest in EMI for testing C compilers [28, 29, 58, 59], 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 [8, 28, 62]. Existing random Simulink model generators include CyFuzz [12] and its successor SLforge [13]. 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.

3 SLEMI: CPS TOOL CHAIN TESTING VIA EMI

Existing EMI-based compiler testing approaches focus on procedural programs [28, 29, 59]. 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 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) [12]. 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., $m \equiv_I n$, 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 are common to both models, as follows.

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

Figure 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* [28, 38]). Then SLEMI performs several one-time base mutations (Listing 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 $m \equiv_I m'$ holds. If via differential testing SLEMI determines that $m \equiv_I m'$ 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.

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 [28, 29, 59], i.e., datatype inference and sample time inference.

3.1.1 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 Figure 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 (*Data Type Conversion* in Simulink) yielding Figure 3b, then Simulink counter-intuitively propagates *int* to $b3$ and $b1$, which is not compatible with $b1$, yielding a compile error.

Listing 1: Base mutations to preprocess seed m using $set(b, p, t)$, which fixes output port p's datatype to t. Besides changing to a fixed-step solver (Section 3.1.2), 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 ∈ 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 ∈ m : // add types
    set(c.b_s, c.p_s, T_dt(c.b_s, c.p_s)) // source output−port
    d := new TypeCast block // for target input−port
    set(d, 0, T_dt(c.b_t, c.p_t)) // d's only output−port (0)
    connect c.b_s → d and d → c.b_t // rewire
```

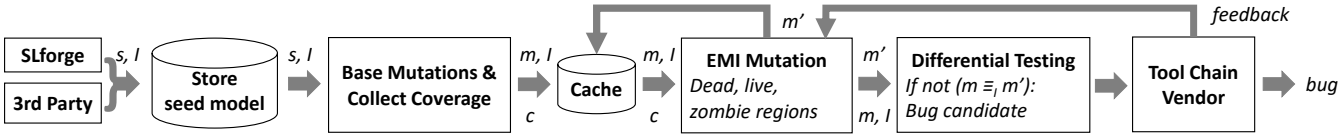

Figure 2: Overview: SLEMI first obtains seed model s with input vector I from a real-world corpus or a random generator (e.g., SLforge), performs one-time base mutations to yield model m, and collects m's coverage c (on I). An EMI-based mutation then yields a valid equivalent (on I) model m' for finding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

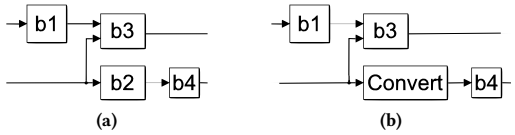

Figure 3: Example seed model excerpt without explicit type specifications (a) where Simulink propagates *double* via $b2$. Replacing nested zombie $b2$ with a *Convert* *Data Type Conversion* block (b) may yield different type inference results.

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[4]) blocks have a parameter that sets the block's output port datatypes. 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 input port datatype.

The lower half of the Listing 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 output-port parameters do not cover all blocks and TypeCast leaves its input-port unspecified. However, this combination has been sufficient and did not create any false warnings in our experiments.

3.1.2 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 [37]). 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.

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

As an extreme example, some blocks directly expose their block-specific sampling frequency, such as *Counter Free-Running*, which plainly returns as its output the number of times it has been sampled [41]. 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 [35] 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 [36]. 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 vary simulation efficiency and 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 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 [14, 36]. 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.

3.2 Mutating Nested Zombie Regions

At a high level this mutation is similar to mutating dead code regions in procedural languages [28]. SLforge-generated models do not use Simulink's default-value inheritance option [40]. In such a model a top-level zombie ignores any values (including defaults) coming from its nested zombie region. For such models this mutation hence changes 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.

Listing 2: Create EMI mutant from preprocessed m.

```
mutate(m, I)
  if randomly chosen block b ∈ 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
```

With the per-block coverage from preprocessing SLEMI identifies nested dynamic zombies. For example, in Figure 1 block *Gain*1 is a nested zombie, as it is nested inside the top-level zombie *Action*1. After deleting a nested zombie block, SLEMI randomly reconnects its predecessor blocks to its successor blocks (top of Listing 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.

3.3 Mutating Zombie and Live Regions

Since prior work has found mutating live code more effective than mutating dead code [59], 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 [29, 59], SLEMI currently focuses on the mutations of the lower part of Listing 2.

First, SLEMI's *live-hierarchy extraction* mutation extracts a live region and promotes it to its own child model (Section 2.1). SLEMI then applies standard Simulink constructs (*Model Reference*) to reference the new model from the original model [41], following modeling rules to preserve equivalence. 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 Figure 4a

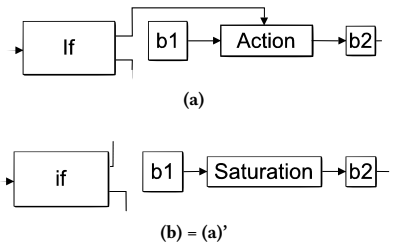

Figure 4: Example live-path mutation (b) that replaces a top-level zombie block (*Action*) in the $b1$ to $b2$ live path with a live *Saturation* block that mimics the default behavior of the replaced top-level zombie block.

example model[5], SLEMI replaces the top-level zombie *Action* block with a live *Saturation* block, yielding Figure 4b. The *Saturation* block constantly feeds the replaced block's default value to its live successor block $b2$.

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.

3.4.1 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* [34]. 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 Figure 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*.

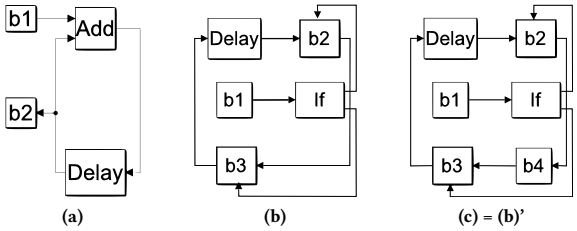

Figure 5: Example benign data dependence loop (a). Adding b_4 to (b) yields an invalid execution order priority in (c).

[5]For brevity Figure 4 and subsequent figures omit the *TypeCast* blocks added by preprocessing.

3.4.2 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 priorities, and language semantics. For example, Simulink orders all block output computations within one block priority level before the blocks of the next level. For example, an *If* block's action subsystems (branches) receive the same exclusive priority. Since the *If* block explicitly controls the execution of all of its action subsytems, they all execute together as an atomic unit whenever the *If* block executes.

This rule disallows placing a block in a path between *If* block's action subsystems. In the valid Figure 5b example model, the $b2$ and $b3$ action subsystems receive the same execution order as that of the *If* block. Adding the $b4$ block outside these subsystems in Figure 5c results in a compile error. Whereas the data dependencies call for $b4$ being computed after $b2$ and before $b3$, blocks $b2$ and $b3$ must execute at the same time as the *If* block, so there is no way to order the execution between $b2$ and $b3$, resulting in an expected compile-time error.

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?

4.1 Seed Models and Their Input Values

The first potential source of seed models is a large corpus of some 1k user-created, publicly available Simulink models [14]. 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 [14]). Only 16 of the corpus models we were able to run had top-level zombie or nested zombie blocks.

In contrast, 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.

As a result, all of our experiments "only" used SLforge-generated (i.e., synthetic) models as seeds for evaluating the research questions. In some sense this is a strength, as using SLforge-generated models enables a fair comparison with SLforge. (Recall that the research questions all revolve around comparing this paper's techniques with SLforge.) Beyond the research questions in the paper, it is too early to tell how this paper's techniques will scale beyond SLforge-generated models.

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 [11].

For our experiments we used the latest SLforge version (in its default configuration) and MATLAB releases R2017a, R2018a, and R2018b [10]. 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*).

4.3 Issue Reporting and MathWorks Feedback

We reported SLEMI-identified issues via MathWorks's Bug Reports website[6] (which required a free MathWorks account). While this website listed earlier bug reports, this list was not comprehensive and did not mention the corresponding technical support case (TSC) numbers. In contrast, open source projects often list all bug reports a project has received, regardless of issue classification and severity. For each report (except one) we eventually received email from MathWorks Support that classified the issue into new/known/non-bug/pending. MathWorks Support usually responded within one business day and interacted further to understand our issue reports. We did not get further feedback from MathWorks Support on their assessment of issue significance or severity.

4.4 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 [47]) 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 [13].

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,

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

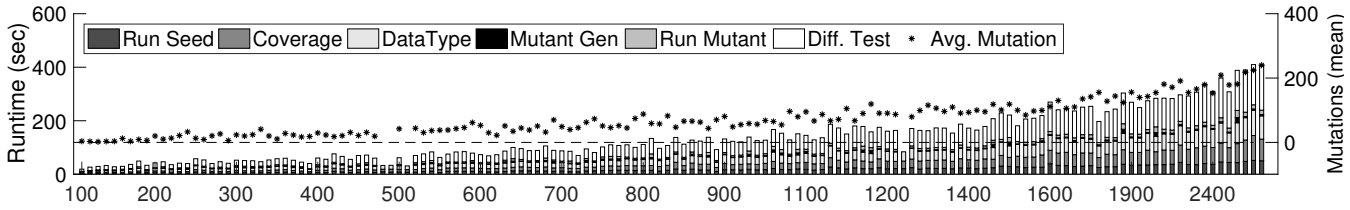

Figure 6: Cumulative runtime of SLEMI's three seed preprocessing and three per-mutant phases; data point = single seed and its mutants; * = average mutation operations that led to a seed's mutants; X-axis = blocks per seed model; data points/bars ordered by x-value and into correct 100-block seed bin, 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).

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.

Figure 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 runtime) 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.

4.5 SLEMI Found New Bugs in Simulink (RQ2)

Table 1 summarizes our reports. To date we have reported 13 unique issues to MathWorks Support, who confirmed 9 as a bug, of which 3 were already known to MathWorks (we re-discovered them independently). Similarly, MathWorks Support told us when they considered a report a non-bug. The corresponding false positive rate was between 2/13 and 4/13 (since 2 cases are still pending).

The mutations that contributed to finding the 9 new/known bugs were the data-type annotation base-mutation (3 bugs), sample-time annotation base-mutation (3), nested-zombie removal (3), top-level zombie removal (1), live fork mutation (1), and live hierarchy mutation (2). (A bug may be impacted by several mutations.) 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.

Case 03205823: Nested Zombie Mutation → Compile Error (Likely New Bug). Figure 7 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 3.1). We independently re-discovered this bug. Simulink R2018a fixed it.

Table 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; *E* = 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	E, T, C
03210493	After mutation Simulink does not eliminate dead *Action Subsystem*	K	E, C
03213776	Valid model stops compiling when collecting inferred properties	N	E, T, C
03259942	Invalid data-dependency loop for *Action* subsystem after mutation; undocumented specification	?	E, S, C
03404633	Output discrepancy for *Discrete Integrator* block due to using different datatype in the mutant	?	E, R
03416784	Output discrepancy for *Sin* block after zero-crossing detection	F	E, R
03475044	Datatype inconsistency after enabling signal logging, due to complex type inference heuristics	N	E, T, C
03486057	When annotating sample time *getSampleTime* API does not return correct sample time	N	E, 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	E, S, C
03489578	Block does not respect sample time inference diagnostic command *InheritedTsInSrcMsg*	K	E, C
03489586	*Signal Editor* block errors-out due to unrelated configuration when specifying sample time	N	E, C
03568445	Behavior difference for Unary Minus operator when feeding small floating-point number	K	E, T, R
03580171	Incorrect sample time inference after live mutation by moving blocks via model-referencing	N	E, C

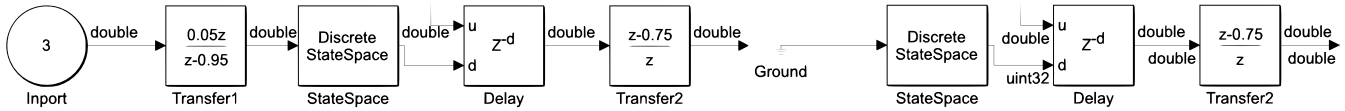

Figure 7: Case #03205823 (condensed) removes dead blocks *Inport* and *Transfer1* in the seed model (left) and replaces them with the type-compatible (double) *Ground* (right) where Simulink failed to infer correct datatypes for all of the blocks, i.e., it inferred uint32 at *Delay*'s input and propagated it back to the output port of *StateSpace* which does not accept it.

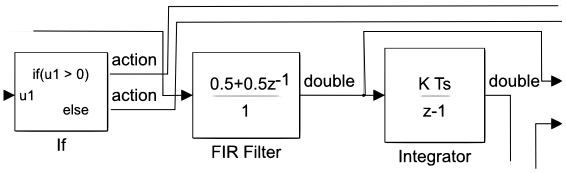

Figure 8: Case #03213776 (condensed): Simulink infers different output types for *FIR FIlter* across different tool chain settings: double in *Normal* mode vs. uint32 when compiling to collect the inferred block properties.

Case 03213776: Nested Zombie Mutation → Mismatch in Different Tool Chain Configurations (Likely New Bug). The excerpted model in Figure 8 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.

4.6 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 9 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 SLEMI initially identified via EMI, if given additional time.

Based on our experience with the SLforge implementation and its differential testing options, the remaining 5 bugs seemed out of reach for SLforge. Specifically, given the Simulink options SLforge uses and the Simulink language constructs it uses in the models it generates, it seems unlikely that SLforge can find these 5 bugs. For example, using sample time analysis and annotation SLEMI discovered a bug (TSC 03489586). SLforge does not perform similar analysis and annotation, so it missed this bug.

4.7 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 [13]), we consider experimenting with other libraries, user-created models and other CPS tools future work.

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.

5.1 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 [46].

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 [39], 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 [61]. 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 [48]. Other recent work transforms Simulink models [16] 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 [55].

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 [64]. To select mutants efficiently, He et al. define an equivalence relation on models [25]. 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.

5.2 EMI-based Mutation for Procedural Code

To complement existing schemes for differential compiler testing, recent work has developed EMI-based mutation for C programs [28, 29, 59, 60] and OpenCL programs [30]. Overall these approaches have found in production-level compilers hundreds of previously unknown bugs, many of which the compiler developers have already fixed [58]. While early EMI-based mutation work focused on mutating dynamically dead program regions through both dead element removal [28] and addition [29], recent work also mutates live program paths [59].

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 publicly available full formal specification and (b) has several key features not found in C or OpenCL, such as explicit notions of time, datatype inference, and zombie code.

5.3 Finding CPS Development Tool Bugs

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 [12] and SLforge [13]. Closely related to SLforge is a random model and differential testing tool by Nguyen et al. [45]. The tool first generates random hybrid automaton models and then HyST [4] translates the automaton models to a variety of CPS modeling languages including Simulink.

Other testing [20, 51, 56, 57] and analysis [20] schemes target selected parts of a CPS development tool. For example, Stürmer et al. test optimization rules of code generators utilizing graph grammars [56, 57]. Fehér et al. model the data-type inferencing logic

of Simulink blocks [20]. In subsequent work, we have implemented one more novel EMI-based mutation technique finding one more Simulink bug (in version R2018a), which is available in Shafiul Azam Chowdhury's doctoral dissertation [9].

5.4 Finding Bugs in CPS Models

Finally, while this paper looks for compiler bugs in CPS tools, a complementary line of work analyzes and looks for bugs in CPS models [2, 3, 7, 22, 27, 31, 39, 53, 65]. For example, MathWorks's Simulink Design Verifier [39] 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 [7] 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 [6, 18, 21, 42, 54], e.g., via mutation testing of Simulink models [6]. Other related work synthesizes controllers that are correct by design [1, 50]. 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.

6 CONCLUSIONS

Finding bugs in commercial cyber-physical system development tools (or "model-based design" tools) such as MathWorks's 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 (CPS) 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 closely related CPS modelling languages, including Stateflow [24] and the Simulink/Stateflow subset TargetLink [19].

ACKNOWLEDGMENTS

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

[2] Rajeev Alur. 2011. Formal verification of hybrid systems. In *Proc. 11th International Conference on Embedded Software, (EMSOFT) 2011*. ACM, 273–278. https://doi.org/10.1145/2038642.2038685

[3] Rajeev Alur, Aditya Kanade, S. Ramesh, and K. C. Shashidhar. 2008. Symbolic analysis for improving simulation coverage of Simulink/Stateflow models. In *Proc. 8th ACM & IEEE International Conference on Embedded Software (EMSOFT)*. ACM, 89–98. https://doi.org/10.1145/1450058.1450071

[4] Stanley Bak, Sergiy Bogomolov, and Taylor T. Johnson. 2015. HYST: A Source Transformation and Translation Tool for Hybrid Automaton Models. In *Proc. 18th International Conference on Hybrid Systems: Computation and Control (HSCC)*. ACM, 128–133.

[5] Olivier Bouissou and Alexandre Chapoutot. 2012. 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)*. ACM, 129–138. https://doi.org/10.1145/2248418.2248437

[6] Angelo Brillout, Nannan He, Michele Mazzucchi, Daniel Kroening, Mitra Purandare, Philipp Rümmer, and Georg Weissenbacher. 2010. Mutation-Based Test Case Generation for Simulink Models. In *Formal Methods for Components and Objects: 8th International Symposium, FMCO 2009, November 4-6, 2009. Revised Selected Papers*. Springer, 208–227.

[7] Lennon C. Chaves, Iury Bessa, Lucas C. Cordeiro, Daniel Kroening, and Eddie Batista de Lima Filho. 2017. Verifying digital systems with MATLAB. In *Proc. 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 388–391.

[8] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An Empirical Comparison of Compiler Testing Techniques. In *Proc. 38th International Conference on Software Engineering (ICSE)*. ACM, 180–190.

[9] Shafiul Azam Chowdhury. 2019. *Automated Testing of a Commercial Cyber-Physical System Development Tool Chain*. Ph.D. Dissertation. University of Texas at Arlington.

[10] Shafiul Azam Chowdhury et al. 2018. SLforge web site. https://github.com/verivital/slsf_randgen/wiki. Accessed Jan 2020.

[11] Shafiul Azam Chowdhury et al. 2020. SLEMI web site. https://github.com/shafiul/slemi. Accessed Jan 2020.

[12] 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. https://doi.org/10.1007/978-3-319-51738-4_4

[13] 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.

[14] 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.

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

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

[17] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust Typechecker Using CLP (T). In *Proc. 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 482–493. https://doi.org/10.1109/ASE.2015.65

[18] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. 2008. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27, 7 (July 2008), 1165–1178. https://doi.org/10.1109/TCAD.2008.923410

[19] dSPACE Inc. 2020. TargetLink. https://www.dspace.com/en/inc/home/products/sw/pcgs/targetli.cfm. Accessed Jan 2020.

[20] Péter Fehér, Tamás Mészáros, László Lengyel, and Pieter J. Mosterman. 2013. Data type propagation in Simulink models with graph transformation. In *Proc. 3rd Eastern European Regional Conference on the Engineering of Computer Based*

Systems (ECBS-EERC). IEEE, 127–137. https://doi.org/10.1109/ECBS-EERC.2013.24

[21] Kamran Ghani, John Clark, and Yuan Zhan. 2009. Comparing algorithms for search-based test data generation of MATLAB Simulink models. In *Proc. IEEE Congress on Evolutionary Computation*. 2940–2947.

[22] Antoine Girard, A. Agung Julius, and George J. Pappas. 2008. Approximate Simulation Relations for Hybrid Systems. *Discrete Event Dynamic Systems* 18, 2 (2008), 163–179. https://doi.org/10.1007/s10626-007-0029-9

[23] Christoph Guger, Alois Schlogl, Christa Neuper, Dirk Walterspacher, Thomas Strein, and Gert Pfurtscheller. 2001. Rapid prototyping of an EEG-based brain-computer interface (BCI). *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 9, 1 (March 2001), 49–58.

[24] Grégoire Hamon and John Rushby. 2007. An operational semantics for Stateflow. *International Journal on Software Tools for Technology Transfer* 9, 5 (2007), 447–456. https://doi.org/10.1007/s10009-007-0049-7

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

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

[27] Aditya Kanade, Rajeev Alur, Franjo Ivancic, S. Ramesh, Sriram Sankaranarayanan, and K. C. Shashidhar. 2009. Generating and Analyzing Symbolic Traces of Simulink/Stateflow Models. In *Proc. 21st International Conference on Computer Aided Verification (CAV)*. Springer, 430–445. https://doi.org/10.1007/978-3-642-02658-4_33

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

[29] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *Proc. 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 386–399.

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

[31] Bing Liu, Lucia, Shiva Nejati, and Lionel C. Briand. 2017. Improving fault localization for Simulink models using search-based testing and prediction models. In *Proc. 24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE.

[32] MathWorks Inc. 2018. Products and Services. http://www.mathworks.com/products/. Accessed Jan 2020.

[33] MathWorks Inc. 2020. About Data Types in Simulink. https://www.mathworks.com/help/simulink/ug/working-with-data-types.html. Accessed Jan 2020.

[34] MathWorks Inc. 2020. Algebraic Loop Concepts — MATLAB & Simulink. https://www.mathworks.com/help/simulink/ug/algebraic-loops.html. Accessed Jan 2020.

[35] MathWorks Inc. 2020. Blocks for Which Sample Time is Not Recommended — MATLAB & Simulink. https://www.mathworks.com/help/simulink/ug/sampletimehiding.html. Accessed Jan 2020.

[36] MathWorks Inc. 2020. Choose a Solver — MATLAB & Simulink. https://www.mathworks.com/help/simulink/ug/types-of-solvers.html. Accessed Jan 2020.

[37] MathWorks Inc. 2020. Sample Time — MATLAB & Simulink. https://www.mathworks.com/help/simulink/sample-time.html. Accessed Jan 2020.

[38] MathWorks Inc. 2020. Simulink Coverage — MATLAB & Simulink. https://www.mathworks.com/help/slcoverage/. Accessed Jan 2020.

[39] MathWorks Inc. 2020. Simulink Design Verifier — MATLAB & Simulink. https://www.mathworks.com/products/sldesignverifier.html. Accessed Jan 2020.

[40] MathWorks Inc. 2020. Simulink Documentation — Conditional Subsystem Initial Output Values. https://www.mathworks.com/help/simulink/ug/specify-or-inherit-conditional-subsystem-initial-values.html. Accessed Jan 2020.

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

[42] Reza Matinnejad, Shiva Nejati, Lionel C. Briand, and Thomas Bruckmann. 2016. SimCoTest: A test suite generation tool for Simulink/Stateflow controllers. In *Proc. 38th International Conference on Software Engineering, (ICSE)*. ACM, 585–588. https://doi.org/10.1145/2889160.2889162

[43] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107. http://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf

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

[45] Luan Viet Nguyen, Christian Schilling, Sergiy Bogomolov, and Taylor T. Johnson. 2015. Runtime Verification of Model-based Development Environments. In *Proc. 15th International Conference on Runtime Verification (RV)*.

[46] Bentley James Oakes. 2014. *Optimizing Simulink Models*. Technical Report SOCS-TR-2014.5. McGill University.

[47] Marta Olszewska, Yanja Dajsuren, Harald Altinger, Alexander Serebrenik, Marina A. Waldén, and Mark G. J. van den Brand. 2016. Tailoring complexity metrics for Simulink models. In *Proc. 10th European Conference on Software Architecture Workshops*.

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

[49] 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.

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

[51] Prahladavaradan Sampath, A. C. Rajeev, S. Ramesh, and K. C. Shashidhar. 2007. Testing Model-Processing Tools for Embedded Systems. In *Proc. 13th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 203–214. https://doi.org/10.1109/RTAS.2007.39

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

[53] Steve Sims, Rance Cleaveland, Ken Butts, and Scott Ranville. 2001. Automated validation of software models. In *Proc. 16th International Conference on Automated Software Engineering (ASE)*. IEEE, 91–96.

[54] Adepu Sridhar, D. Srinivasulu, and Durga Prasad Mohapatra. 2013. Model-based test-case generation for Simulink/Stateflow using dependency graph approach. In *Proc. 3rd IEEE International Advance Computing Conference (IACC)*. 1414–1419. https://doi.org/10.1109/IAdCC.2013.6514434

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

[56] Ingo Stürmer and Mirko Conrad. 2003. Test suite design for code generation tools. In *Proc. 18th IEEE International Conference on Automated Software Engineering (ASE)*. 286–290. https://doi.org/10.1109/ASE.2003.1240322

[57] Ingo Stürmer, Mirko Conrad, Heiko Dörr, and Peter Pepper. 2007. Systematic Testing of Model-Based Code Generators. *IEEE Transactions on Software Engineering (TSE)* 33, 9 (Sept. 2007), 622–634. https://doi.org/10.1109/TSE.2007.70708

[58] Zhendong Su and Chengnian Sun. 2020. EMI Compiler Validation Project. http://web.cs.ucdavis.edu/~su/emi-project/. Accessed Jan 2020.

[59] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding Compiler Bugs via Live Code Mutation. *SIGPLAN Not.* 51, 10 (Oct. 2016), 849–863. https://doi.org/10.1145/3022671.2984038

[60] Qiuming Tao, Wei Wu, Chen Zhao, and Wuwei Shen. 2010. An Automatic Testing Approach for Compiler Based on Metamorphic Testing Technique. In *Proc. 2010 Asia Pacific Software Engineering Conference (APSEC)*. IEEE, 270–279.

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

[62] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 283–294. https://doi.org/10.1145/1993498.1993532

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

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

[65] Liang Zou, Naijun Zhan, Shuling Wang, and Martin Fränzle. 2015. Formal Verification of Simulink/Stateflow Diagrams. In *Proc. 13th International Symposium on Automated Technology for Verification and Analysis (ATVA)*. Springer, 464–481.