Automatically Finding Bugs in a Commercial Cyber-Physical System Development Tool Chain With SLforge

Shafiul Azam Chowdhury
University of Texas at Arlington
Arlington, Texas

Soumik Mohian
University of Texas at Arlington
Arlington, Texas

Sidharth Mehra
University of Texas at Arlington
Arlington, Texas

Taylor T. Johnson
Vanderbilt University
Nashville, Tennessee

Christoph Csallner
University of Texas at Arlington
Arlington, Texas

ABSTRACT

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

To address these shortcomings, we created the first large collection of public Simulink models and used the collected models' properties to guide random model generation. To further guide model generation we systematically collected semi-formal Simulink specifications. In our experiments on several hundred models, the resulting SLforge generator was more effective and efficient than the state-of-the-art tool CyFuzz. SLforge also found 8 new confirmed bugs in Simulink.

ACM Reference format:
Shafiul Azam Chowdhury, Soumik Mohian, Sidharth Mehra, Taylor T. Johnson, and Christoph Csallner. 2018. Automatically Finding Bugs in a Commercial Cyber-Physical System Development Tool Chain With SLforge. In *Proceedings of ICSE, Gothenburg, Sweden, May 2018 (ICSE'18),* 12 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Cyber-physical system developers rely heavily on complex development environments or tool chains, which they use to design graphical *models* (i.e., *block-diagrams*) of cyber-physical systems. Such models enable engineers to do rapid prototyping of their systems through simulation and code generation [23]. Since automatically generated native code from these data-flow models are often deployed in safety-critical environments, it is crucial to eliminate bugs from cyber-physical system tool chains [8, 40].

Ideally, one should formally verify such tool chains, since a tool chain bug may compromise the fidelity of simulation results or introduce subtle bugs in generated code [9]. However, a commercial cyber-physical system (CPS) development tool chain consists of millions of lines of code, so formal verification does not (yet) scale to such tool chains. While compilers and other CPS tool chain components remain mostly unverified, we continue to observe frequent safety recalls in various industries [1, 55, 56]. The recalls are attributed to hidden bugs in the deployed CPS artifacts themselves, in spite of spending significant efforts in their design validation and verification [6, 57].

Testing, on the other hand, is a proven approach to effectively discover defects in complex software tool chains [35]. Especially *randomized differential testing* has recently found over a thousand bugs in popular production-grade compilers (e.g., GCC and LLVM) that are part of CPS development tool chains [17, 25, 31, 45, 58]. The technique eliminates the need of a test-oracle and can hammer the *system under test* in the absence of a complete formal specification of the system under test—a phenomenon we commonly observe in commercial CPS tool chain testing [7, 24, 50]. Differential testing seems suitable for black-box testing of the entire CPS tool chain, and its most susceptible parts (e.g., code generators) in particular [43, 50]. *CyFuzz* is the first (and only) known randomized differential testing tool for CPS data-flow languages [11].

While CyFuzz initiated the work for testing CPS tool chains, more work is necessary to evaluate the scheme's capabilities, e.g., for finding bugs in Simulink that developers care about. For instance, a random model generator should generate tests with properties similar to the models people typically use, since they are more likely to get fixed by the system under test (SUT) developers. While large repositories of publicly available programs of various procedural and object-oriented programming languages exist [12, 22, 59], we are not aware of such a collection of CPS models. The existing CPS studies rely on a handful of public [41] or proprietary [47] Simulink models.

Among other shortcomings, the models CyFuzz generates are small and lack many syntactic constructs. Recent studies identified expressive test-input generation as a success-factor for compiler validation ([31, 58]). Perhaps due to its inability to generate large tests with rich language features, CyFuzz has not found previously unknown bugs. Furthermore, CyFuzz essentially generates *invalid* models and iteratively fixes them until the SUT can compile and simulate them without error or exception. However, this heuristic approach required several time-consuming iterations and did not use Simulink specifications, which are available publicly in natural language.

To address these shortcomings, we have conducted the first study of a large number of public Simulink models. The size of many of

ICSE'18, May 2018, Gothenburg, Sweden
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

these models is larger than the average size of models used in industry. From the collected models we obtain properties that are useful for targeting a random Simulink model generator. Our model collection is publicly available and may eliminate the nontrivial overhead of artifact-collection in future studies.

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

Finally, we provide the first approach to *Equivalent modulo input* (EMI) testing in CPS development tool testing [31]. SLforge creates EMI variants from the random models it generates and uses them in the differential testing setup. During an approximately five months long testing time, we found and reported 12 bugs overall, MathWorks confirmed 10 of them, of which 8 were previously unknown. To summarize, the paper makes the following major contributions.

- To better target a random CPS model generator, we conduct the first large-scale study of publicly available Simulink models. A significant portion of these models are of size and complexity that are comparable to models used in industry.
- We identify problems in the existing CPS random model generator CyFuzz and design solutions that directly led to the discovery of new bugs in the Simulink tool chain.
- Finally, by comparing it with CyFuzz, we evaluate SLforge's efficiency and bug-finding capability.

2 BACKGROUND

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

2.1 CPS Data-flow Models And Simulink

While in-depth descriptions are available elsewhere [54], the following are the key concepts. In a CPS development tool (e.g., Simulink), a user designs a CPS as a set of dataflow *models*. A model contains *blocks*. A block accepts data through its *input ports*, typically performs on the data some operation, and may pass output through its *output ports* to other blocks, along *connection* lines. Simulink specifies which port (of a block) supports which *data-types*.

In a connection, we name the block sending output *source* and the block receiving data a *target*. Output ports are numbered starting with 1. Input port numbering starts with 0, where 0 denotes a special port (e.g., the *Action* port of the *If Action* block). In addition to such explicit connections, using *From* and *Goto* blocks, one can define implicit (*hidden*) connections [40, 44].

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

More formally, block $b \in B$ and connection $c \in C$ may be part of model $m \in M$. Then a flat (non-hierarchical) model is a tuple $\langle B, C \rangle$ where $m.B$ and $m.C$ denote the model's blocks and connections.

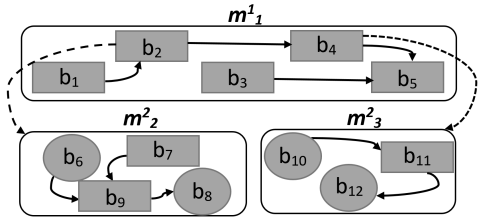

Figure 1: Example hierarchical CPS model: Rounded rectangle = model; shaded = block; oval = I/O; solid arrow = dataflow; dashed arrow = hierarchy.

Each connection is a tuple $\langle b_s, p_s, b_t, p_t \rangle$ of source block b_s, source output port p_s, target block b_t, and target input port p_t. While a Simulink connection may have multiple targets, we break such a multi-target connection into multiple (single-target) connection tuples, without losing expressiveness.

For hierarchical models we list a model m^i at hierarchy level i with its n direct child models as $m^i[m_k^{i+1}, \ldots, m_{k+n-1}^{i+1}]$. The Figure 1 example $m_1^1[m_2^2, m_3^2]$ has m_1^1 as its *top-level* model. m_2^2 and m_3^2 are m_1^1's *child* models at hierarchy level 2. The dashed arrow starting at b_2 indicates that in the m_1^1 model b_2 is a *placeholder* for the m_2^2 model. Block b_1 sends data to m_2^2, where b_6 receives it. Block b_8 sends data back to b_4 in m_1^1.

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

After designing a model in Simulink, users typically *compile* and *simulate* it. In simulation, Simulink numerically solves the model's mathematical relationships established by the blocks and their connections and calculates various non-dead (Section 4.3) block's outputs according to user-requested sample-times and inferred context-dependent *time steps*, using built-in *solvers* [34]. Simulink offers different *simulation modes*. While in *Normal* mode Simulink "only" simulates blocks, it also emits some code for blocks in *Accelerator* mode, and a standalone executable in *Rapid Accelerator* mode.

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

2.2 Testing Simulink With CyFuzz

CyFuzz is the first known differential testing framework for CPS tool chains [11]. The framework has five phases. The first three phases create a random model and the last two phases use the model to automatically test a SUT.

Specifically, starting from an empty model, (1) the *Select Blocks* phase chooses random blocks and places them in the model. (2) The *Connect Ports* phase connects the blocks' ports arbitrarily, yielding a model the SUT may reject, e.g., due to a type error. For example, an

output port's data type may be incompatible with the data type of the input port it connects to. (3) CyFuzz iteratively fixes such bugs in the *Fix Errors* phase, by responding to the SUT's error messages with corresponding repair actions. This "feedback-driven model generation" approach, despite being an imperfect heuristic, can fix many such model errors.

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

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

2.3 EMI-based Compiler Testing

Equivalent modulo input (EMI) testing is a recent advancement in differential testing of compilers for procedural languages [31]. Complementing plain differential testing, EMI found over one hundred bugs in GCC and LLVM [9]. The idea is to systematically mutate a source program as long as its semantics remain equivalent under the given input data. Engineering basic mutators is relatively easy and the overall scheme can effectively find bugs, when combined with a powerful random generator that can create expressive test inputs (e.g., Csmith [58]).

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

3 PUBLIC SIMULINK MODEL COLLECTION

To understand the properties of CPS data-flow models designed by both researchers and engineers, we conducted the first large study of Simulink models. The largest earlier Simulink model collection we are aware of contains some 100k blocks [16]. However, these models are company-internal and thus not available for third-party studies. In contrast, our collection consists of some 145k blocks, which are all publicly available (some require a standard Simulink license as they are shipped with Simulink).

For context, earlier work reports that at Delphi, a large industrial Simulink user, an average Simulink model consists of several hundred blocks [33]. Of the models we collected, 35 consist of more than 1,000 blocks, which is larger than an average model at Delphi.

3.1 Model Collection and Classification

For this study, we used the Simulink configuration our organization has licensed, which includes the base Simulink tool chain and a large set of libraries. This configuration includes the latest Simulink version; the project web page lists the detailed configuration [4]. However, with this configuration, we could directly compile only just over half of the collected models, as the remaining ones required additional libraries that were not part of our configuration. Our (anonymized) project web page contains a detailed list of the collected models [4].

3.1.1 Tutorial. This group consists of official Simulink tutorial models from MathWorks[1]. We manually reviewed the models and their descriptions in the Automotive, Aerospace, Industrial Automation, General Applications, and Modeling Features categories and excluded what we considered toy examples. We also included here the Simulink-provided domain-specific library we had access to, i.e., Aerospace. An example model from this group is *NASA HL-20* (1,665 blocks), which models "the airframe of a NASA HL-20 lifting body, a low-cost complement to the Space Shuttle orbiter" [53].

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

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

As an example, a model from the *Grid-Connected PV Array* project is "a detailed model of a 100-kW array connected to a 25-kV grid via a DC-DC boost converter" created by a senior engineer at Hydro-Quebec Research Institute (IREQ) [42]. It has 1,320 blocks. We classified it as Advanced.

3.1.3 Other. This group consists of models we obtained from academic papers (5 models), the academic research of colleagues (7 models), and Google searches (16 models). An example is the *Benchmarks for Model Transformations and Conformance Checking* released by engineers at Toyota Technical Center California [27]. It has 208 blocks.

3.2 Model Metrics

In this study, we focus on those model properties that are relevant for constraining a random model generator to models that are representative of realistic CPS models. Our Matlab-based tool we used to collect the following metrics is freely available on the project

[1] https://www.mathworks.com/help/simulink/examples.html
[2] https://www.mathworks.com/matlabcentral/fileexchange

Table 1: Overview of collected public models: Total number of models (M); models we could readily compile without extra effort (C); hierarchical models (H); total number of blocks and connections.

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

site [4]. The collected metric values are shown as box-plots with min-max whiskers.

3.2.1 Number of Blocks and Connections. Blocks and connections are the main elements of Simulink models and are counted widely [33, 39]. We have included the contents of masked blocks [54] in the parent model's count. Next, we count the total number of blocks and connections at a particular hierarchy level up to hierarchy level 7.

Our connection-count metric does not include hidden connections. For connections with multiple target ports, we count the connections' target ports. Perhaps not surprisingly, Simple models are smaller (and Advanced models are larger) than models of the other groups (Figures 2a and 2b), since we manually reviewed and classified them in this class.

3.2.2 Hierarchy Depth. Since industrial models are frequently organized as a hierarchy, we measured how deep these hierarchies are. We treated both *subsystems* and *model reference* blocks as adding a hierarchy level. Most of the collected models are indeed hierarchical (i.e., 362/391 models). But the median maximum hierarchy depth did not extend five across all model groups.

More surprising were the distribution of blocks and connections across hierarchy levels (Figures 3a and 3b). These numbers were rather similar across hierarchy levels. Overall, the number of blocks and connections in each hierarchy level were small, as denoted by the small median value.

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

Figure 4 suggests that only a small portion of the blocks are custom ("User_Defin"). Across all four groups, *Ports & Subsystems* and *Math Operations* were the two libraries used most frequently. SLforge thus supports these libraries (among others, see Section 4.1.1), and automatic custom block generation. We also noted a high contribution from the *Signal Routing* library using *From* and *Goto* blocks, which enables establishing hidden data-flow relationship (Section 2.1).

3.2.4 Requested Simulation Duration. This metric captures the total simulation time *requested* by a given model (not the actual CPU time spent in simulating it). Most of the models (except those

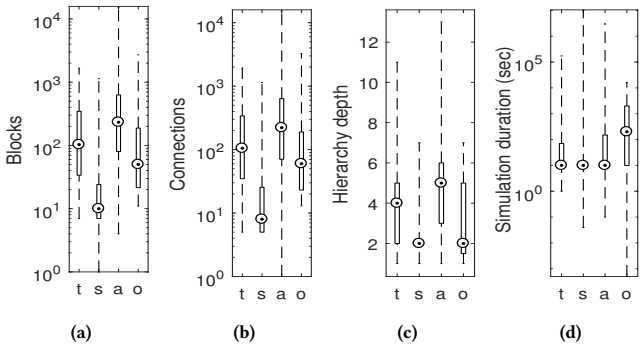

Figure 2: Collected public models: Total blocks (a), connections (b), maximum hierarchy depth (c), and requested simulation duration (d).

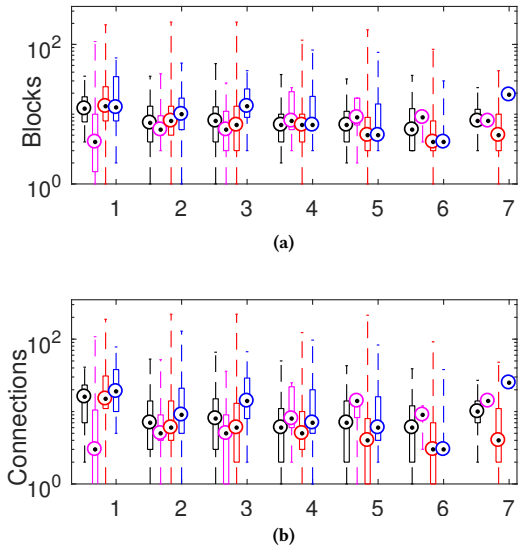

Figure 3: Collected public models: Blocks (a) and connections (b) by hierarchy-level, grouped by model group (t, s, a, and o).

from the Other group) used the default simulation duration value of 10 seconds (Figure 2d). Consequently, we ran simulations using this default value in our experiments, and have not experimented with other possible values yet.

4 SLFORGE

To address the shortcomings in the state-of-the-art differential testing framework for CPS tool chains, this section describes the design of SLforge. Figure 5 gives an overview of the SLforge's seven main phases.

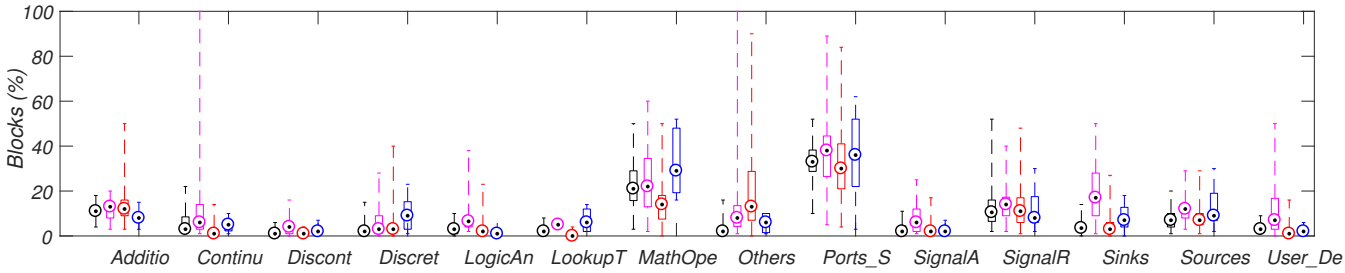

Figure 4: Collected public models: Distribution of blocks across libraries (shortened to the first 7 letters), each from left to right: Tutorial, Simple, Advanced, and Other.

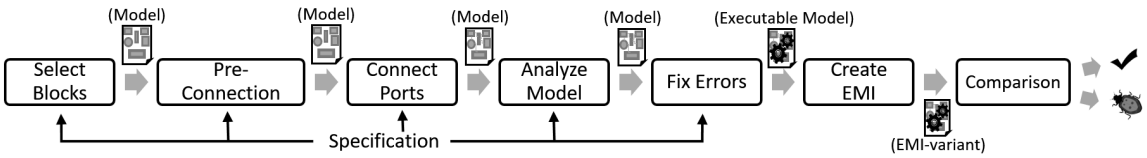

Figure 5: Overview of SLforge's main phases. For space, we merged CyFuzz's last two phases into the single *Comparison* phase.

4.1 Gathering Semi-Formal Specifications

CyFuzz heavily relies on its Fix Errors phase to repeatedly compile and simulate a model and iteratively repair errors based on the content of Simulink-provided error messages. Instead of this time-consuming iterative process, SLforge aims at generating a valid model in the first place, if given the language specifications. Of course, the challenge (which motivated CyFuzz's iterative process) is that there exists no complete, up to date, formal Simulink specification.

4.1.1 Design Choice. Simulink specifications are available as informal and semi-formal descriptions of Simulink behavior, mainly from the various Simulink web sites. From our experiments with CyFuzz, we hypothesized that many of the iterations in the Fix Errors phase are due to block data-type inconsistency and fixing algebraic loops. Besides, in our CyFuzz experiment (Section 5.1) the most frequent error was block *sample time* inconsistency. We collected specifications to both address these issues and to enable creating large hierarchical models (as Simulink users prefer this modeling choice).

So far, we have collected data-type support and block-parameter specifications for all built-in libraries. Other language specifications (Section 4.2) are often block and library specific. Since collecting the entire Simulink language specification would be overwhelming, we collected specifications for blocks from the most-used libraries. Concretely, SLforge supports blocks from *Math Operations*, *Ports and Subsystems*, *Discrete*, *Continuous*, *Logic and Bit Operations*, *Sinks* and *Sources* libraries. This list also covers the CyFuzz-supported libraries and thus helps ease evaluating SLforge.

4.1.2 Collection Process. Using little engineering effort, SLforge's regular expression based parser parsed block data-type and parameter specifications for all built-in blocks. However, due to the limitation of the parser and Simulink's free-form specification style, SLforge can only collect parts of some specification. E.g., for three different ports of the *Variable Time Delay*, *Discrete Filter* and *Delay* blocks, the Direct Feed-through property (Section 2.1) is described as "Yes, of the time delay (second) input", "Only when the leading numerator coefficient does not equal zero" and "Yes, when you clear Prevent direct feedthrough" respectively [54].

4.2 Use of Semi-Formal Specifications

Since complete and updated formal specifications for Simulink are not publicly available, existing work relies on a subset of Simulink operational specifications, which are manually crafted and possibly outdated [7, 24]. Unlike these approaches, we explored collecting specifications directly from official Simulink documentations automatically, using easy-to-engineer parsers. Such parsers can automatically update the collected specifications when new versions of the SUT are released, given the structure of the specifications go under minor or no change. From our experience with recent versions of Simulink specifications (R2015a-R2017a), the specifications SLforge collects indeed had minor structural changes.

Although SLforge parses specifications automatically and stores them using internal data-structure, for the aid of discussion, we introduce a few notions in this section. Extending the notation of Section 2.1, function T_b returns a block's Simulink *block-type*. For example, for each Simulink *If* block, T_b returns If. Next, the *valid* predicate indicates if the Simulink type checker and runtime system accept a given model as legitimate, i.e., when there are no compile or run-time exceptions. Now we can express (part of) the Simulink specification as a formula or specification *rule*. Given such a rule $\delta \in \Delta$, we denote with $m \vDash \delta$ that model m *satisfies* (i.e., complies with) the rule. We observe that a valid model satisfies all (collected) specification rules ($\forall \delta \in \Delta : valid(m) \rightarrow m \vDash \delta$).

4.2.1 Select Blocks Phase. To support built-in libraries, SLforge uses specifications in this phase. Specifically, SLforge-generated models satisfy the following rules by construction. For example,

using usual set cardinality notation, Eq. (1) ensures that for each *If* block, the model has two *IfAction* subsystem blocks (one each for the *if* and *else* branch).

$$2 * |\{ b_1 \in m.B : T_b(b_1) = If \}| = |\{ b_2 \in m.B : T_b(b_2) = IfAction \}| \quad (1)$$

By parsing Simulink documentation, SLforge obtains a set S of blocks from the *Sinks* and *Source* libraries that are only valid in the top-level model, as enforced by Eq. (2). Similarly, Eq. (3) restricts using illegitimate blocks in non-top-level models, depending on the hierarchy-type property of the model (Section 2.1), using predicate *supports*. The predicate holds only when model m^i's hierarchy-type (first argument) allows block b in m^i, based on b's block-type (second argument).

$$\forall b \in m^i.B: (i > 1) \rightarrow (T_b(b) \notin S) \quad (2)$$

$$\forall b \in m^i.B: (i > 1) \rightarrow supports(T_h(m^i), T_b(b)) \quad (3)$$

$$\forall b \in m.B: ((T_h(m) \in W) \wedge stime(b) = stime(driver(m))) \vee (T_h(m) \notin W \wedge st(T_b(b), b)) \quad (4)$$

Eq. (4) configures each block's sample time property *stime*. When used in hierarchical model m of a W-listed hierarchy-type, block b's sample time should match the sample time of the model's driver (Section 2.1). In all other cases, we use predicate *st*, which holds only when the sample time property of block b is properly configured according to its block-type. To enforce such rules, SLforge propagates information from parent to child models.

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

$$\forall c \in m.C: (T_b(c.b_t) = IfAction) \wedge (c.p_t = 0) \rightarrow (T_b(c.b_s) = If) \quad (5)$$

$$\forall c \in m.C: (T_b(c.b_s) = If) \rightarrow (T_b(c.b_t) = IfAction) \wedge (c.p_t = 0) \wedge |\{ c_2 \in m.C : c_2.b_s = c.b_s \wedge c_2.p_s = c.p_s \}| = 1 \quad (6)$$

Eq. (5) and Eq. (6) together specify the control-flow from an *If* block to its *If Action* blocks. Specifically, each *If* block output port is connected to a single (Eq. (6)) *If Action* block *Action* port.

4.2.3 Analyze Model Phase. On the current model state, SLforge now removes algebraic loops and assigns data-types. Instead of querying a *disjoint-set* data structure every time SLforge connects two blocks to detect whether connecting them will create a cycle, we detect them later in this phase using a single graph traversal *remove_algebraic_loops* (Listing 1) on each of the child models and on the top-level model. In contrast, CyFuzz relies on Simulink built-in functions to fix algebraic loops; SLforge discovered a previously unknown Simulink bug in these features (Section 5.3.1). Specifically, SLforge identifies *back-edges* and interrupts them with *Delay* blocks [13]. Since this process changes m, SLforge ensures that the model remains valid. For example, to ensure that the model satisfies the rules in Eq. (5) and Eq. (6), instead of placing a *Delay* block between an *If* and an *If Action* block, Listing 1 places it before the *If* block.

Listing 1: Removing possible algebraic loops from a model. color(b) denotes a block's visit-status via do_dfs method: white=unvisited; gray and black: visited.

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

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

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

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

4.3 EMI-testing

As recent work suggests that EMI-testing is promising for compiler testing [31], we explored this direction for Simulink. EMI-testing for Simulink could take many forms. For example, one could extend a model with blocks and connections that remain dead under the existing inputs. As another example, one could statically remove some of the dead blocks.

In this work we infer an EMI-variant from a given randomly generated model, by removing all blocks that are dead. We approximate the set of dead blocks statically, using Simulink's *block reduction* feature [54]. This approach differs from the original EMI implementation ([31]) in the sense that we collect the dead-block information statically, while [31] dynamically collected code coverage

information. We chose the static approach as it required minimal engineering effort.

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

4.4 Classification of Bugs

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

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

4.4.2 Essential Feature. Here we discuss bugs based on the *essential* generator/differential testing feature that helped discovering them. We attribute *Hierarchy* to a bug if SLforge can reproduce the bug only by creating hierarchical models. Next, intuitively, SLforge attributes *Specification* to a bug when it identifies Simulink specification mismatches. Finally, like CyFuzz, SLforge identifies *Comparison* bugs by simulating a model varying SUT options (Section 2.2). As a special case, SLforge attributes *EMI* to a bug if some EMI-variant of a successfully simulated model does not compile, or results in comparison error when after-simulation signal data of the EMI-variant is compared with the original model or with other EMI-variants.

5 EVALUATION

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

- **RQ1** Can SLforge generate models systematically and efficiently in contrast to CyFuzz?
- **RQ2** Can SLforge generate feature-rich, hierarchical models in contrast to CyFuzz?
- **RQ3** Can SLforge effectively test Simulink to find bugs in the popular development tool chain?

To answer these research questions, we implemented SLforge on top of the open-source CyFuzz implementation. In the evaluation, we ran SLforge on 64-bit Ubuntu 16.04 virtual machines (VM), of 4 GB RAM and 4 processor cores each. We have used two identical host machines (Intel i74790 CPU (8 cores) at 3.60 GHz; 32 GB RAM each). When measuring runtime and other performance metrics (RQ1), we ran SLforge on each of the otherwise idle host machines (one VM per host). To find bugs (RQ3) we ran up to five VMs on each of these two hosts.

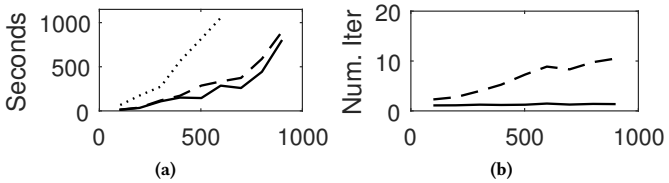

Figure 6: Runtime on valid models by model size given in blocks: (a) Average runtime of model generation; (b) Average number of required iterative fixes. Solid = SLforge; dashed = SLforge without specification usage and analyses; dotted = CyFuzz.

5.1 SLforge Generates Models More Systematically and Efficiently (RQ1)

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

We compared the average time taken to generate a valid model (i.e., from Select Blocks to Fix Errors, inclusively). We also measured the number of iterative model-correction steps (*Num. Iter.*) in the Fix Errors phase. However this metric was not available in Exp.CF. In each of these experiments we started with generating 100 blocks (on average) per model, and gradually increased the average number of blocks (by 100 in each step, using 20 models in each step), up to 800 blocks/model on average. To approximate the bug-finding capability of these three setups, we counted the total number of unique bugs found in each of these experiments. Data in Both SLforge versions had a lower average runtime than CyFuzz (Figure 6a).

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

Similarly, SLforge needs fewer iterations in the Fix Errors phase (Figure 6b) in Exp.S+. Moreover, this value remains almost constant in Exp.S+. However, perhaps not surprisingly, the number increases with the number of blocks in Exp.S-. This result indicates that SLforge generates models more systematically as it reduces the dynamic error-connection steps significantly.

Next, we examine how the changes in SLforge affect the tool's bug-finding capability. The total number of unique bugs found in Exp.S+, Exp.S- and Exp.CF are 4, 1, and 0, respectively. Exp.S+ found the same bug discovered in Exp.S-, and found 3 more bugs. While investigating, we observed that having specifications enabled SLforge finding those bugs, since without the specifications, SLforge

could not determine whether it should report a bug given an error message returned by Simulink. As an example, consider compiling a model with Simulink which results in a data-type inconsistency error between two blocks in the model. Leveraging the data-type support specifications of the two blocks, SLforge can report a bug when it does not expect any data-type inconsistency between the blocks.

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

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

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

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

5.3 SLforge Found New Bugs in Simulink (RQ3)

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

We have reported almost all of the SLforge-suspected bugs to MathWorks except two cases where we had associated the bug to an implementation error. For each of the reported cases, MathWorks has indicated if it considers the case a bug. For this work we mark a report as a *false positive* if MathWorks considers the case a non-bug. Our rate of false-positive is low: 2/12 reports.

Table 2 summarizes all the bugs we have reported. MathWorks has confirmed 10 of our reported issues as unique bugs, of which 8 are new bugs. Following are details of a representative subset of the confirmed bugs, including SLforge-generated models we manually reduced to fit the space and aid bug-reporting. Automated test-case reduction is part of future work. These models are freely available [4].

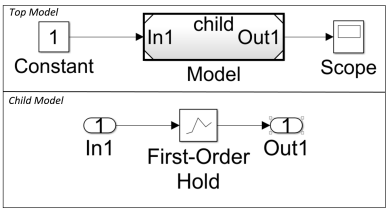

Figure 7: Bug TSC-02472993: The *Model* block (top) refers to the child model (bottom). Simulink fails to handle rate transition automatically, leading to a runtime failure.

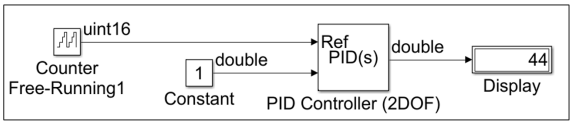

Figure 8: Bug TSC-02386732: While specified to only accept *double* inputs, Simulink does not raise a type error for this *PID Controller (2DOF)* accepting a *uint16*.

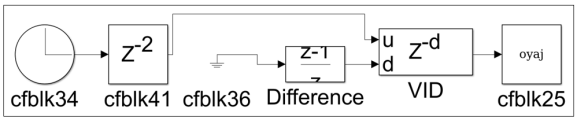

Figure 9: Bug TSC-02614088: In spite of supporting *double* data-type in its *d* port, block VID issued a data-type consistency as it *prefers integer* type.

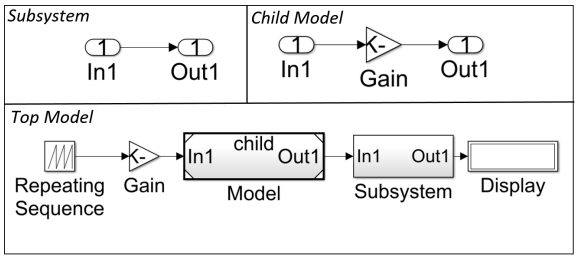

Figure 10: Bug TSC-02515280: For the child model (top right) Simulink's *Verification and Validation* toolbox API calculates inconsistent *SubSystemCount* values. MathWorks ruled the API specification ambiguous, as it did not properly define the API's scope.

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

5.3.2 Specification bug. After incorporating Simulink specifications into various SLforge phases, SLforge started to identify bugs caused by specification violation. For example, bug TSC-02472993

Table 2: SLforge-discovered issues and confirmed Simulink bugs: *TSC* = *Technical Support Case* number from MathWorks; *St* = status of bug report (*NB* = new bug, *KB* = known bug, *FP* = false positive); *P* = discovery point (*C* = Compile Time, *R* = Runtime); *F* = bug type based on Essential Feature (*A* = Hang/Crash Error, *S* = Specification, *C* = Comparison, *H* = Hierarchy, *E* = EMI, ? = not further investigated); *Ver* = Latest Simulink version affected.

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

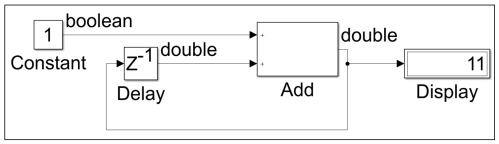

Figure 11: Issue TSC-02382873: When using the *internal rule* to detect the *Add* block's output data-type, the rule fails to choose a correct data-type for the second input port (e.g. *double*) and throws a compilation error.

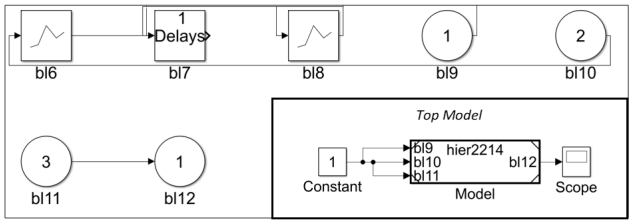

Figure 12: EMI bug TSC-02476742: The *Top Model*'s (in bottom right corner) *Model* block is a placeholder for the child model (top and left), where all blocks except *bl11* and *bl12* are dead.

of Figure 7 manifests when Simulink fails to handle blocks operating at different sample times, leading to runtime failure which is not expected as per specification. The bug only occurs for the *First-Order Hold* block and when SLforge generates hierarchical models. As another example, Figure 8 depicts Simulink's *PID Controller (2DOF)* block accepting data of type unsigned int, whereas the specification states that the block only accepts data of type double (TSC-02386732).

In another case we noted that in spite of supporting type *double* in port *d*, block *Variable Integer Delay* (block *VID* in Figure 9) resulted in type-mismatch error. After reporting the issue, MathWorks suggested that the port "prefers" integer types and thus issued a type mismatch error when it was given a double type. This specification is not publicly available. Lastly, the Figure 10 issue (TSC-02515280) MathWorks classified as expected behavior, where Simulink's count of the number of *Subsystems* did not match our count. However, part of Simulink's results are inconsistent and the specification has been found ambiguous, resulting in a new confirmed bug.

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

5.3.4 EMI bug. Figure 12 illustrates Simulink bug TSC-02476742. Notice how only block bl11 is connected to an Outport block bl12, hence all remaining child blocks are dead and can be removed in EMI-testing. While EMI-testing in Normal mode removed all dead child nodes, EMI-testing in Accelerator mode failed to do so, which MathWorks classified as a Simulink bug.

6 DISCUSSION

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

7 RELATED WORK

In the following we discuss the most closely related work.

7.1 Studies of CPS Models and Their Properties

Empirical studies of real-wold programs date back at least to the 1970s [30], and have gained increasing interest due to the wide availability of open source programs [52]. Several pieces of earlier work compute useful properties from open source Java programs [12, 22, 59]. For example, the RUGRAT random Java program generator is configured based on such metrics [26].

Tempero et al. presented the *qualitas corpus*—a curated collection of Java programs to evaluate studies that use Java source code as subjects [52]. Although similar work has been performed in other domains [10, 36, 51], we are not aware of related work in the CPS domain, which is significantly different than traditional procedural or object-oriented programming languages.

Recent studies introduced measures for Simulink model modularity [15] and complexity [39], but only evaluated the result using a limited number of models. Compared to their works, we perform the first large-scale study on more than 391 Simulink models. Consequently, similar to [52], our collection of artifacts may serve as a corpus for Simulink-model based empirical studies.

7.2 Differential Testing

Recent work has found many compiler bugs using differential testing. To generate programs that are syntactically correct, many of the test generators harness the language's context-free grammar and a pre-determined probability table from which the generator chooses grammar elements [17, 25]. To generate programs that are also well-typed, McKeeman imposes the type information directly onto the *stochastic grammar* the generator uses [35].

Csmith, on the other hand, uses various analysis and runtime checks to generate programs with no undefined behavior [58]. Other techniques generate well-typed programs using knowledge of the type-system of the underlying language (e.g., JCrasher for Java [14]) and using constraint-logic programming (such as the Rust typechecker fuzzer [17]).

Whereas all these approaches target compilers of traditional textual languages (including procedural, object-oriented, and functional ones), they do not address the challenges inherited in testing CPS development environments [11]. CyFuzz pioneered the work for differentially-testing CPS development tools, but the prototype for Simulink was ineffective in finding new bugs. In contrast, this work addresses CyFuzz's limitations by incorporating the informal Simulink specifications in the random model generation process. Besides increasing efficiency, we can generate larger models with rich language features that led to finding new bugs.

Recent work complements randomized differential testing via EMI-testing. For example, Le et al. hammer C language compilers [31]. We harness the technique to create EMI-variants of Simulink models for the first time. Lidbury et al. discuss the effectiveness of randomized differential testing and EMI for OpenCL compilers [32] and [9] performs a comprehensive empirical compiler testing evaluation.

Among other works, Nguyen et al. present a runtime verification framework for CPS model analysis tools leveraging random hybrid automata generation [28, 38]. In contrast, our generator does not rely on model transformations [5], which may limit the efficiency of existing work [38].

Other testing schemes target parts of the CPS tool chain. For example, Stürmer et al. test optimization rules of code generators utilizing graph grammars [49, 50]. However, complete and updated formal specifications for most commercial CPS development tools are unavailable and such white-box testing in parts was found undesirable [46].

Sampath et al. discuss testing CPS model-processing tools using semantic *Stateflow* meta-models [46]. Unfortunately, the approach does not scale and updated specifications are unavailable, due to the rapid release cycles of commercial CPS tools [17, 49]. Fehér et al. model the data-type inferencing logic of Simulink blocks for reasoning and experimental purposes [19]. While these works focus on a small part of the entire CPS tool chain, we differentially-test the entire CPS tool chain harnessing the available informal (but updated) specifications.

7.3 Test Case Generation for Simulink Models

SLforge is loosely related to test case generators for existing Simulink models [8, 18, 37]. While both research directions target the CPS domain, existing test case generators find bugs in Simulink models while SLforge finds bugs in Simulink itself.

Liu et al. improve fault localization of Simulink models using search-based techniques [33]. Ghani et al. compare search-based techniques for test data generation [20]. With similar objectives, related work generates test inputs for Simulink models [21, 34, 48].

Finally, the literature is also rich in verification and formal analysis of CPS models [2, 47, 60]. E.g., Alur et al. generate symbolic traces from Simulink models and analyze coverage improvements [3, 29]. None of these works, however, generates random tests for the Simulink tool chain itself to discover bugs in it, in comparison to our contributions in this paper.

8 CONCLUSIONS

This paper identified and addressed key challenges of random differential testing of the popular CPS development environment Simulink. To generate random tests for Simulink that have properties similar to those of public models, we conducted the first large-scale study of publicly available Simulink models. Our model collection may help compare other empirical studies and reduce the nontrivial overhead of collecting Simulink models.

This paper also identified and addressed other shortcomings of the existing state-of-the-art Simulink differential testing scheme CyFuzz, namely not using existing informal Simulink specifications and not generating large, feature-rich hierarchical models. Using parsed Simulink specifications and various analyses, SLforge generates models efficiently compared to CyFuzz and finds more bugs. Furthermore, this paper explored incorporating equivalence modulo input (EMI) based testing in the CPS domain. In total, we discovered two existing and 8 new confirmed Simulink bugs.

REFERENCES

[1] H. Alemzadeh, R. K. Iyer, Z. Kalbarczyk, and J. Raman. 2013. Analysis of Safety-Critical Computer Failures in Medical Devices. *IEEE Security Privacy* 11, 4 (July 2013), 14–26. https://doi.org/10.1109/MSP.2013.49

[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] Anonymous Authors. 2017. Project Homepage. https://github.com/AutoTestGen/Simulink/wiki. (2017).

[5] 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.

[6] Boris Beizer. 1990. *Software testing techniques* (second ed.). Van Nostrand Reinhold.

[7] 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

[8] 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, Eindhoven, The Netherlands, November 4-6, 2009. Revised Selected Papers*, Frank S. de Boer, Marcello M. Bonsangue, Stefan Hallerstede, and Michael Leuschel (Eds.). Springer, 208–227.

[9] 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.

[10] R. J. Chevance and T. Heidet. 1978. Static Profile and Dynamic Behavior of Cobol Programs. *SIGPLAN Not.* 13, 4 (April 1978), 44–57.

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

[12] Christian S. Collberg, Ginger Myles, and Michael Stepp. 2007. An empirical study of Java bytecode programs. *Softw., Pract. Exper.* 37, 6 (2007), 581–641. https://doi.org/10.1002/spe.776

[13] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. 2001. *Introduction To Algorithms*. MIT Press.

[14] Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: An automatic robustness tester for Java. *Software—Practice & Experience* 34, 11 (Sept. 2004), 1025–1050. https://doi.org/10.1002/spe.602

[15] Yanja Dajsuren, Mark G.J. van den Brand, Alexander Serebrenik, and Serguei Roubtsov. 2013. Simulink Models Are Also Software: Modularity Assessment. In *Proc. 9th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*. ACM, 99–106.

[16] Florian Deissenboeck, Benjamin Hummel, Elmar Juergens, Michael Pfaehler, and Bernhard Schaetz. 2010. Model Clone Detection in Practice. In *Proc. 4th International Workshop on Software Clones (IWSC)*. ACM, 57–64.

[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] V. D'Silva, D. Kroening, and G. 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] P. Fehér, T. Mészáros, L. Lengyel, and P. J. Mosterman. 2013. Data Type Propagation in Simulink Models with Graph Transformation. In *2013 3rd Eastern European Regional Conference on the Engineering of Computer Based Systems*. 127–137. https://doi.org/10.1109/ECBS-EERC.2013.24

[20] K. Ghani, J. A. Clark, and Y. Zhan. 2009. Comparing algorithms for search-based test data generation of Matlab (R) Simulink (R) models. In *2009 IEEE Congress on Evolutionary Computation*. 2940–2947. https://doi.org/10.1109/CEC.2009.4983313

[21] 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

[22] Mark Grechanik, Collin McMillan, Luca DeFerrari, Marco Comi, Stefano Crespi, Denys Poshyvanyk, Chen Fu, Qing Xie, and Carlo Ghezzi. 2010. An Empirical Investigation into a Large-scale Java Open Source Code Repository. In *Proc. ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, 11:1–11:10.

[23] C. Guger, A. Schlogl, C. Neuper, D. Walterspacher, T. Strein, and G. 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] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proc. 21th USENIX Security Symposium*. USENIX Association, 445–458. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

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

[27] Xiaoqing Jin, Jyotirmoy V. Deshmukh, James Kapinski, Koichi Ueda, and Ken Butts. 2017. Benchmarks for Model Transformations and Conformance Checking. https://cps-vo.org/node/12108. (2017). Accessed May 2017.

[28] Taylor T. Johnson, Stanley Bak, and Steven Drager. 2015. Cyber-physical specification mismatch identification with dynamic analysis. In *Proc. ACM/IEEE Sixth International Conference on Cyber-Physical Systems (ICCPS)*. ACM, 208–217. https://doi.org/10.1145/2735960.2735979

[29] 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

[30] Donald E. Knuth. 1971. An Empirical Study of FORTRAN Programs. *Softw., Pract. Exper.* 1, 2 (1971), 105–133.

[31] 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.

[32] 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.

[33] B. Liu, Lucia, S. Nejati, and L. 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)*. 359–370.

[34] 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

[35] 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

[36] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of Unix Utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44. https://doi.org/10.1145/96267.96279

[37] M. Mohaqeqi and M. R. Mousavi. 2016. Sound Test-Suites for Cyber-Physical Systems. In *10th International Symposium on Theoretical Aspects of Software Engineering (TASE)*. 42–48. https://doi.org/10.1109/TASE.2016.33

[38] 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)*.

[39] 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 *Proccedings of the 10th European Conference on Software Architecture Workshops, Copenhagen, Denmark, November 28 - December 2, 2016*. 5. http://dl.acm.org/citation.cfm?id=3004853

[40] 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* (2017), 1–23. https://doi.org/10.1007/s10009-017-0450-9

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

[42] Pierre Giroux. 2017. Grid-Connected PV Array - File Exchange - Matlab Central. http://www.mathworks.com/matlabcentral/fileexchange/34752-grid-connected-pv-array. (Aug. 2017).

[43] A. C. Rajeev, Prahladavaradan Sampath, K. C. Shashidhar, and S. Ramesh. 2010. CoGenTe: A tool for code generator testing. In *Proc. 25th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 349–350. https://doi.org/10.1145/1858996.1859070

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

[45] Jesse Ruderman. 2007. Introducing jsfunfuzz. https://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/. (2007).

[46] 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

[47] S. Sims, R. Cleaveland, K. Butts, and S. Ranville. 2001. Automated validation of software models. In *Proc. 16th Annual International Conference on Automated Software Engineering (ASE)*. 91–96. https://doi.org/10.1109/ASE.2001.989794

[48] A. Sridhar, D. Srinivasulu, and D. P. 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

[49] 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

[50] 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

[51] Giancarlo Succi, Witold Pedrycz, Snezana Djokic, Paolo Zuliani, and Barbara Russo. 2005. An Empirical Exploration of the Distributions of the Chidamber and Kemerer Object-Oriented Metrics Suite. *Empirical Software Engineering* 10, 1 (2005), 81–104.

[52] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. 2010. The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *2010 Asia Pacific Software Engineering Conference*. 336–345.

[53] The MathWorks Inc. 2017. NASA HL-20 Lifting Body Airframe - Matlab & Simulink. https://www.mathworks.com/help/aeroblks/nasa-hl-20-lifting-body-airframe.html. (2017). Accessed May 2017.

[54] The MathWorks Inc. 2017. Simulink Documentation. http://www.mathworks.com/help/simulink/. (2017). Accessed May 2017.

[55] U.S. Consumer Product Safety Commission (CPSC). 2010. Recall 11-702: Fire Alarm Control Panels Recalled by Fire-Lite Alarms Due to Alert Failure. http://www.cpsc.gov/en/Recalls/2011/Fire-Alarm-Control-Panels-Recalled-by-Fire-Lite-Alarms-Due-to-Alert-Failure. (Oct. 2010).

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

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

[58] 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

[59] Hongyu Zhang and Hee Beng Kuan Tan. 2007. An Empirical Study of Class Sizes for Large Java Systems. In *Proc. 14th Asia-Pacific Software Engineering Conference (APSEC)*. 230–237.

[60] 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)*, Bernd Finkbeiner, Geguang Pu, and Lijun Zhang (Eds.). Springer, 464–481.