# Runtime Verification of Cyber-Physical System Development Environments (Term Paper)

Shafiul Azam Chowdhury
The University of Texas at Arlington
Arlington TX 76019, USA
Email: shafiulazam.chowdhury@mavs.uta.edu

Siddhant Gawsane
The University of Texas at Arlington
Arlington TX 76019, USA
Email: siddhant.gawsane@mavs.uta.edu

Sidharth Mehra
The University of Texas at Arlington
Arlington TX 76019, USA
Email: sidharth.mehra@mavs.uta.edu

***Abstract*—Over the years, cyber-physical system (CPS) development environments have facilitated the development of complex systems, by simulating their functionalities and automatically generating deployable artifacts. Bugs that might occur in a CPS development environment, can therefore cause expensive cost overheads at the very least and critical failures at the most. Differential testing has been proven to be of great merit in identifying bugs in these emulation environments. Cyfuzz[1] attempts to apply the concept of differential testing to try find bugs in Cyber-physical systems, specifically the Simulink program available as part of the Matlab ecosystem. Our work is a study of open source Simulink models from a variety of sources, their metrics and complexity parameters, and how they compare to those of the models generated by Cyfuzz.**

## I. Introduction

Model-based design of cyber-physical system (CPS) heavily relies on complex development environments or tool chains, which are used to design graphical *models* (aka *block-diagrams*) of CPS. Such models enable engineers, researchers and students to do rapid prototyping of their systems through simulation and code generation [2]. Since automatically generated native code from these *data-flow* models often find themselves deployed in safety-critical environments, it is crucial to eliminate bugs from the entire CPS tool chain. Ideally one should formally verify such development environments to ensure that some hidden bug in the complex tool chain does not compromise the fidelity of simulation or code-generation process. However, formally verifying an entire CPS development tool chain which consists millions of lines of code is yet unfortunately, not scalable.

Testing, on the other hand, is a proven approach to effectively discover defects in complex software tool chain [3]. One particular sort of testing, randomized differential testing (aka *fuzz-testing*) has recently found more than thousands of bugs in popular production-grade compilers (e.g. GCC, LLVM), which are part of CPS development tool chain [4], [5], [6], [7], [8]. Fuzz-testing eliminates the necessity of a test-oracle and can hammer the system under test *(SUT)* in the absence of complete formal specification of the SUT - a phenomenon we commonly observe in commercial CPS tool chain testing [9], [10], [11].

Effectiveness of recent compiler validation projects elicited interest in investigating the applicability of such techniques in popular CPS development environment (e.g. MathWorks' Matlab/Simulink) testing, resulting in the inception of the *CyFuzz* project [1], [12]. CyFuzz laid the foundation for applying randomized differential testing for arbitrary CPS data-flow language and published an open source implementation to test Simulink. Differential testing scheme seems quite suitable for black-box testing of an entire CPS development tool chain, and its most susceptible parts (e.g. code generators) in particular [9], [13].

Although CyFuzz initiated the work for fuzz-testing any CPS tool chain which adheres a conceptual modeling language, and dealt with the unprecedented challenges (e.g. mix of textual and graphical programming paradigms, missing complete and updated formal semantics of the SUT etc.), more work is necessary to evaluate its full capabilities in finding bugs in Simulink. In particular, the models which CyFuzz generate are small in size and lack essential syntactic constructs, hence limiting the expressiveness of generated tests. Perhaps due to its inability to generate large tests with many language features, CyFuzz did not find new bugs in the original experiments. A random model generator should generate models which represents real-world models at the very least.

In our investigation, we did not find any study on the CPS models people use in the real-world. In particular, we were interested in *complexity* and other useful properties of Simulink models. Indeed, since a random model generator for CPS models were not available, there were simply not enough motivation to conduct such studies which have already been conducted in other domains, empirical study on Java programs and byte-code, for instance [14], [15], [16]. Many of the properties of CPS models available in the public domain can be used to drive a random model generator to generate block-diagrams which are representative of real-world models.

Empirical study on existing CPS models has two-fold benefits: besides finding its usage in random model generation, such model properties answer various interesting questions. For example, how complex (structurally and in terms of numerically simulating the model) are the models people use in real world or how large are they? How are the models organized (flat layout vs. hierarchical organization)? In general, what are the properties one should look at when performing such an empirical study on CPS models? Statically collected information in this process can also be useful to validate claims

which defines and correlates such metrics to useful factors (understandability of models, e.g.) [17], [18].

First section of the paper conducts an empirical study on wide variety of Simulink models from different sources which are available in the public domain. We identify and obtain interesting properties from the models, which we call *model metrics*. We then extended CyFuzz's capabilities to generate random models taking some of these metrics as configuration parameters. CyFuzz essentially generates invalid[1] models and iteratively updates the model so that the SUT compiler accepts it. However, this heuristic-based approach required several time-consuming iterations to fix a model and did not make use of the available specifications. Incorporating available (informal) specifications in the model-generation process, we generate models more efficiently and found bugs where implementation in the SUT did not match its specification.

Besides enhancing the model generator, We explored applicability of *equivalent modulo input (EMI)*-based testing, a new, proven direction in differential testing of programming language compilers [6]. Using a *mutator*, EMI-based approaches take a random program and generates many equivalent (modulo input) programs from it and leverages all of them to fuzz-test the SUT. Engineering mutators does not take much efforts but the overall scheme can effectively find bugs in the presence of random generator which creates expressive test inputs.

Since recent fuzz-testing schemes identified expressive test-input generation as success-factor, we extend CyFuzz's capabilities and present a new tool *RandGen* to generate models with advanced Simulink language features [5]. RandGen also creates EMI variants from the random models it generate and use them in the differential testing setup, eventually finding a new bug in the SUT. In a short period of testing time, RandGen found 7 confirmed, previously unknown bugs in total, along with other issues in the popular CPS development tool chain.

To summarize, we make following contributions in this paper:

- To understand various properties of CPS models, we collect and study (Simulink) models available in the public domain. We identify various properties of interest and present a visualization of the data we collected (Section III).
- We extend existing random model generator CyFuzz and introduce improvements which directly led to the discovery of new bugs in the Simulink tool chain. Section IV illustrates our design choices and how we addressed interesting challenges in the process.
- Finally, Section V evaluates our work by answering interesting research questions.

## II. Background Materials

### A. CPS data-flow models and Simulink

Commercial CPS development environments (e.g. Simulink) enable its users to design their systems using data-flow *models*. A model consists of many *blocks*; each block can accept data using *input ports* and may pass data to some other blocks using its *output ports*. We call the line between an input and an output port a *connection*. CPS development environments offer many built-in blocks which are organized in *libraries*. Commercial data-flow languages provide wide selection of libraries. Besides using built-in blocks, one can directly use native code (e.g. C/Matlab in Simulink) to implement custom functionalities for a block; we refer to such blocks as *custom blocks*.

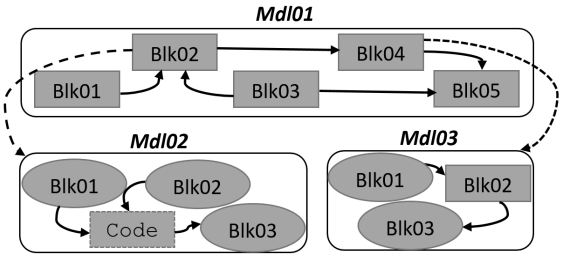

Fig. 1: Illustrative CPS model

As an example, we present three hypothetical CPS models in Fig. 1. In our example, rounded rectangles denote models, rectangles denote blocks and solid lines denote connections, where the arrow indicates direction of data flow. We use *x::y* notation to unambiguously identify some block *x* in some model *y*. Dashed lines (with arrows) are used to design models in hierarchies. Origin of a dashed line is a block which is placeholder for some model pointed by the arrow. Ovals in the figure denotes special blocks using which a model can exchange data with other models.

In our example, *Mdl01* is the *top model* (hierarchy level 1), whereas *Mdl02* and *Mdl03* are *child* models of *Mdl01* (both are in hierarchy level 2). *Mdl01::Blk02* is a placeholder for *Mdl02* in *Mdl01*, as indicated by the dashed arrow. *Mdl01::Blk01* passes some data to *Mdl02*; inside the child model *Blk01* accepts this data [2]. In Section V-C, we present screen-shots of some actual Simulink models which reproduce the bugs RandGen discovered.

Although in our notations a child model is an individual model, this is not always the case in commercial data-flow languages. For example, in Simulink one can create hierarchy using `subsystem` or `model reference` blocks. In our example, *Mdl01::Blk02* and *Mdl01::Blk04* could be any of these two types of blocks. One significant difference between these two block-types is that `model reference` block creates an *instance* of the model it is referencing to. The referenced model is an independent model by itself which resides in a separate file in the file system and may be independently simulated. The model can be reused in same or different unrelated models by simply creating an instance of it. In contrast, `subsystems` share the same file space with their parent model and can not be independently simulated, hence restricting its reuse.

[1] models which are rejected by the SUT compiler

[2] *Mdl02::Blk02* could also accept the data, this will be unambiguously defined in an actual CPS modeling language

Simulink also enables users to model control-flow using `If`, `For Iterator` blocks etc. While Simulink allows some types of cycles in the graph representation of Simulink models involving the connection lines (solid arrows in Fig. 1), cycles in hierarchies (dashed arrows) are undesirable.

After designing a model in Simulink, users typically *compile* and then *simulate* it. Simulink numerically solves the mathematical relationships established in the model and calculates output for its blocks at various *time steps* using *solvers* [19]. The time-steps are typically determined by the solver and usually vary for different types of blocks, based on their *sample time* property.

Simulink offers different *simulation modes*; the default mode is called *Normal* and Simulink usually, does not generate code for its blocks in this mode. On the contrary, Simulink usually generates code (e.g. in *Accelerator* mode) and even standalone executable from the graphical model in other modes. By changing *targets*, users can instruct Simulink to generate code for desired platforms where the code will be ultimately deployed. For example, the *Embedded coder* product generates efficient code to be deployed in embedded hardware [20].

For a detailed discussion of model-based design and Simulink, refer to [20].

### B. CyFuzz: differential testing framework for Simulink

CyFuzz is the very first differential testing framework for a conceptual CPS modeling language [1]. The framework has five phases; first three phases create a random model, and the last two phases use the model to fuzz-test a SUT. We briefly discuss CyFuzz's phases in this section. Starting with an empty model, the *Select Blocks* phase chooses random blocks and places them in the empty model. *Connect Ports* phase connects ports of blocks arbitrarily, resulting in a model which might be rejected by the SUT. For example, if a block doesn't support the data-type which is passed to it using a connection, then the model will not even compile successfully. CyFuzz iteratively fixes the model in the *Fix Errors* phase, by compiling the model and collecting information from the error message(s) returned by the SUT. This "feedback-driven model generation" approach, in spite of being an imperfect heuristic, can surprisingly fix many of the errors from the random model. See Fig. 2 for an schematic overview of RandGen, which is built on top of CyFuzz. Phases which are borrowed from CyFuzz are not shaded [1].

CyFuzz's *Log Signals* phase phase takes a valid (models which are accepted by the SUT as a legitimate block-diagrams) randomly generated model and simulates it varying different SUT options. The phase records block-outputs (aka *signals*) for each block's output ports at various time-steps for each of the variations of the SUT configuration. CyFuzz's prototype realization for Simulink varies different simulation modes partly to exercise various code generators in the Simulink tool chain (see Section II-A). Using this recorded data, *Compare* phase tries to detect dissimilarity in some block's signal in two different simulation set-ups. As an example, CyFuzz could take *Mdl01* from Fig. 1 and simulate it in *Normal* and *Accelerator* mode. Let the terms $Mdl01 :: Blk02^{Nml}_{t1}$ and $Mdl01 :: Blk02^{Acl}_{t1}$ denote the block's output at $t1$ time step for Normal and Accelerator mode, respectively. If these two terms are not *equivalent*[3], this may indicate a presence of bug. Other types of bugs CyFuzz hunts for are crashes and deteriorated SUT performance.

### C. EMI-based Testing

EMI-based testing is a recent advancement in compiler fuzz-testing [6]. Le et. al. coined the term in their C compiler validation project, where they automatically generated equivalent (modulo input) C-programs from an original program by stochastically removing dead parts from it. The project mainly leverages Csmith to generate random C programs which do not take inputs and thus are suitable for EMI-testing. The random programs are then mutated to derive equivalent programs, which are then used in a differential testing set-up. The technique proved to be very effective by finding multiple bugs in production-grade C compilers (GCC, LLVM).

## III. An empirical study on CPS data-flow models

We collect publicly available CPS data-flow models to study and understand various properties of them. We restrict ourself to Simulink models in this study. We collect metrics which can express complexity of the models, as well as various parameters required by a random model generator to generate models representative of real-world CPS data-flow diagrams. In the following sections, we classify the models based on their source and introduce the metrics of interest.

### A. Model classes

We identify following four sources to collect Simulink models from. Table I summarizes general information on the models we collected and analyzed. Details about the individual models can be found here[4].

- *Examples*: In this class, we include Models from Simulink examples website[5] and models from Simulink's `Aerospace` block-set). We have only used these categories from Simulink examples: *Automotive*, *Aerospace* and *Industrial Automation* applications.
- *Matlab Central*: Matlab Central[6] platform allows users to share their Simulink models and related files (functions, data etc.) which can be readily downloaded. We collected 16 models using the search feature and sorting the results by selecting "Ratings - high to low".
- *GitHub*: GitHub.com[7] is a popular project hosting platform. We used its search feature with keywords "Simulink" and sorted repositories by "Favorites" count in descending order. We further search using file extension matching to identify repositories which contains files with .mdl or .slx extensions.

[3] See [1] for a rigorous discussion of CyFuzz's signal comparison framework
[4] Available: http://bit.ly/2oTPyLF
[5] Available: https://www.mathworks.com/help/simulink/examples.html
[6] Available: https://www.mathworks.com/matlabcentral/fileexchange
[7] Available: https://github.com/

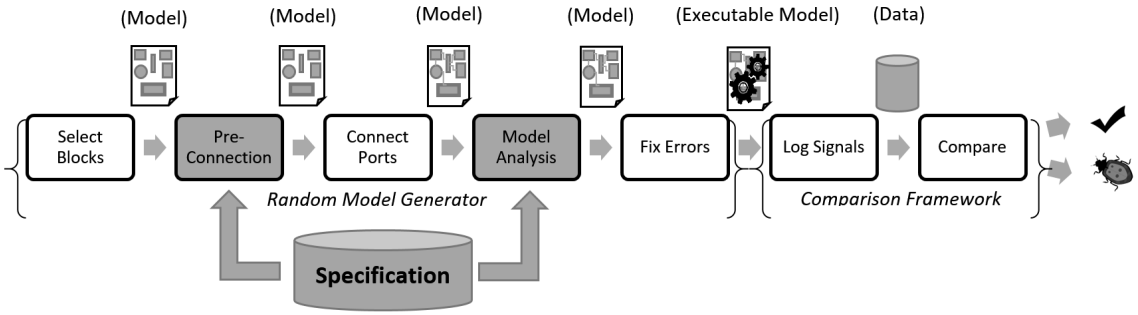


Fig. 2: Schematic overview of RandGen's phases

TABLE I: General information on collected models. *#Models*, *#Compilable* and *#Hierarchy* denotes the total number of models, the number of models which we could compile readily without extra effort and the number of models which have hierarchy respectively.

| Class | #Models | #Compilable | #Hierarchy |
|---|---|---|---|
| Examples | 42 | 41 | 41 |
| GitHub | 162 | 94 | 149 |
| Matlab Central | 52 | 16 | 45 |
| Other | 26 | 13 | 19 |
| Total 282 | 164 | 254 | |

- *Other*: In this class, we include rest of the models, most of which we obtained from academic papers and search engine results. We also enlist models received from our colleagues which they use in academic research in this category.

### B. Data Collection

We collect various information from the public Simulink models which broadly falls into two categories: complexity metrics and configuration parameters required by a random model generator. To collect data and compute metrics, we used a custom tool (written in Matlab and available in the project homepage) which leverages various Matlab APIs. We heavily use box-plots to visualize data skipping some of the outliers due to space limitation. This section introduces the metrics along with visualizations of their representative data, whereas Section V discusses our observation in details.

In this study, we identify following set of metrics to be collected from a Simulink model to investigate:

*1) Number of blocks and connections:* Total number of blocks in the model and total number of connections between them is our first metric of interest, since these metrics denote size of a model. This is also a required parameter for the CyFuzz tool. We have explored `masked` blocks[8] and all the blocks (and connections) inside all child-representing blocks while computing the metrics. We present a visualization of blocks-count in Fig. 3 and connections-count in Fig. 4. We

[8]In Simulink, `masks` hide implementations of a block [20]

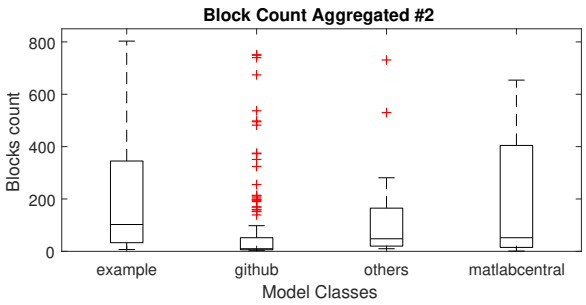


Fig. 3: Number of blocks in a model (aggregated for all levels)

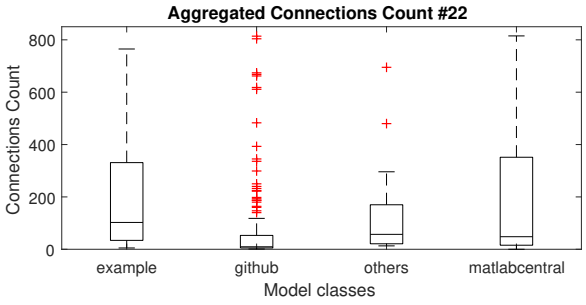


Fig. 4: Number of connections in a model (aggregated for all levels)

also count the number of unique blocks for these four model classes in Fig. 5.

Next, we present hierarchy level-wise count (counting total number of blocks and connections at a particular hierarchy level) for blocks (Fig. 6) and connections (Fig. 7) . We present data up to hierarchy level 5.

grouped by model classes Examples, GitHub, Matlab Central and Other (left to right)

*2) Hierarchy depth:* As organizing models using hierarchical structure is a common practice in model-based design of CPS, we investigated this metric (Fig. 8). In our data-collection, we identified both `subsystems` and `model reference` blocks (see Section II-A) as hierarchy-representing blocks. A random model generator requires parameter similar to "maximum hierarchy depth" to determine when to stop when recursively generating models in some

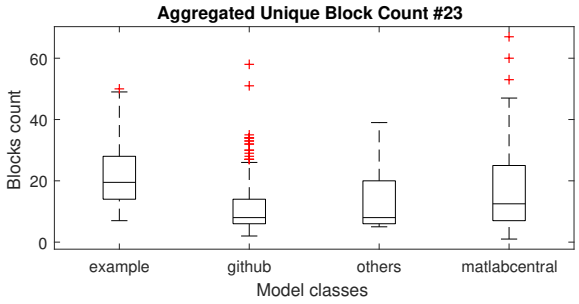


Fig. 5: Unique blocks count (aggregated for all levels)

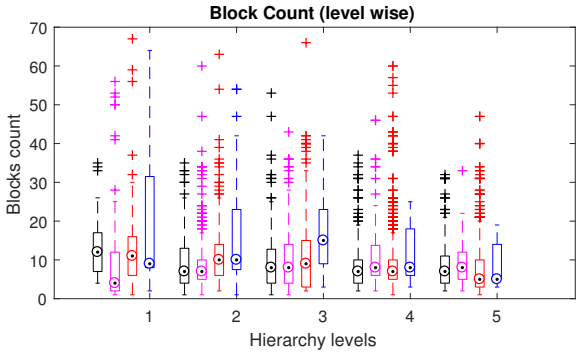


Fig. 6: Blocks count (hierarchy-level wise), grouped by model classes Examples, GitHub, Matlab Central and Other (left to right)

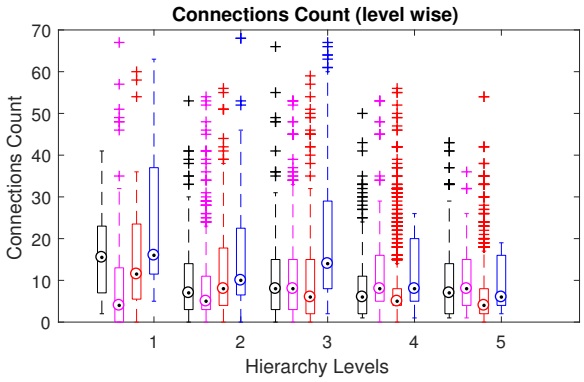


Fig. 7: Connections count (hierarchy-level wise), grouped by model classes Examples, GitHub, Matlab Central and Other (left to right)


Fig. 8: Maximum hierarchy depth

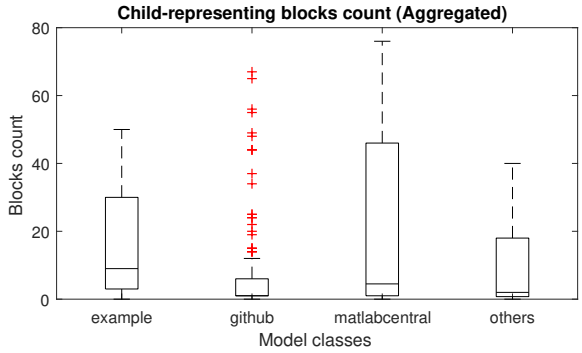


Fig. 9: Count of child-model representing blocks (aggregated for all hierarchy levels)

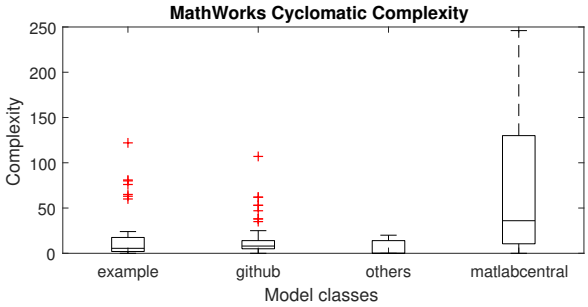


Fig. 10: Cyclomatic complexity of Simulink models

hierarchy.

*3) Number of child-model representing blocks:* For each model, we count number of blocks which denote child-models (Fig. 9). We compute this metric as earlier work relates it to complexity [18] and is also used by random model generator.

*4) Child model reuse:* Commercial CPS development tools allow reusing some existing child models in the same model. However, our study found only one model which reuses its children.

*5) MathWorks cyclomatic complexity:* MathWorks define cyclomatic complexity of an object (e.g. block) by $\sum_{n=1}^{N}(o_n - 1)$, where $N$ is the number of decision points in the object and $o_n$ is the number of outcomes at $n^{th}$ decision point [21]. We used Matlab's `Verification and Validation` toolbox to compute this measure (Fig. 10), noting that for some of the models this API did not yield any result.

*6) Library Participation:* This metric identifies which libraries participate in model creation. For each of the blocks in the model, we identified its associating library. In the cases where we fail to identify the library (due to Matlab API limitations), we assign the block under imaginary library *other*. Fig. 11 presents the distributions of blocks in libraries for model classes Examples, GitHub and Matlab Central respectively. We exclude Other models due to space limitations.

*7) Compilation Time:* For the models which we could compile readily, we record the compilation time. This metric can represent complexity of the model. Fig. 12 presents the

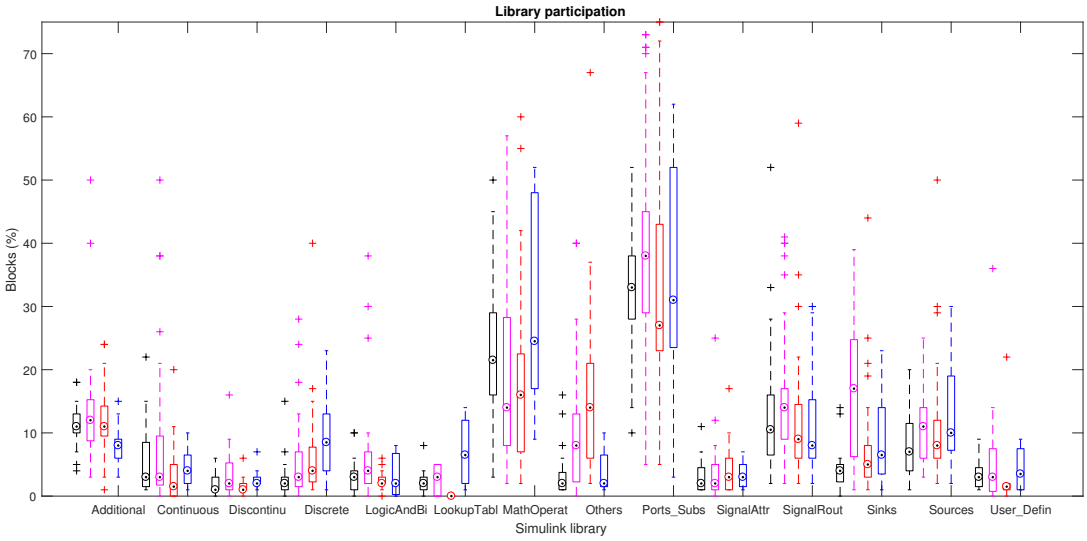


Fig. 11: Participation of blocks from libraries, grouped by model classes Examples, GitHub, Matlab Central and Other (left to right)

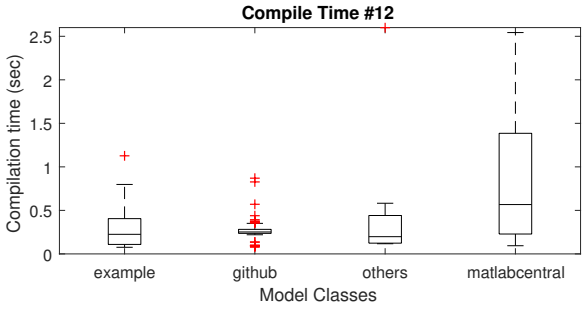


Fig. 12: Compilation time of Simulink models

compilation time required by models for each model class.

*8) Number of connected components:* Feedback-loops present in a model can indicate complexity of a model in terms of how much challenging it is to simulate using a SUT. We calculate number of connected components which is a lower bound for number of feedback loops present in the model. We also identify presence of algebraic loops. Of the 50 only three (one each in Examples, GitHub and Matlab Central class) contains algebraic loop in them.

## IV. Finding Simulink bugs using RandGen

Here we discuss our extension of CyFuzz which found 7 bugs in Simulink during a short experiment period. Although we discuss our methodology in terms of Simulink, the technique is applicable for testing any model-based CPS development environment. Fig. 2 shows the schematic overview of RandGen, which besides introducing two new phases (shaded in gray in Fig. 2, notably enhanced other phases of CyFuzz. One major distinction between the two random-generators is that RandGen uses a database of Simulink specifications which can be used to generate valid models accepted by the SUT. Using these specifications, RandGen's Pre-connection phase fulfills certain requirements entailed by specific blocks. The specifications are again used in the Analyze Model phase later, to remove certain errors from the model before passing it to the Fix Errors phase of CyFuzz.

### A. Using Simulink specifications in random generation

One of our major contributions is that we collect specifications from Simulink documentation and generate models accordingly. Using specification in random generation has two-fold benefit: (a) this helps creating a compilable model without heavily depending on the Fix Errors phase of CyFuzz, and, (b) since models are generated according to specification, RandGen can expose bugs related to incorrect implementation of specifications.

From Simulink documentations, which denotes specifications in semi-structured natural language, we have collected input and output data-type support information for various built-in blocks, numerous parameters they support along with their data-types and other information (e.g. whether block is *direct feed-through*[9]).

### B. EMI-based Testing

As recent work suggests EMI-based testing to be very promising in compiler testing, we explored this direction in this project [6]. Taking a randomly generated model, we generate an equivalent (modulo inputs) model . As RandGen-generated models does not take any inputs, we can readily generate EMI models by removing *dead blocks* from the

[9]A non-direct-feed-through block does not depend on its inputs when computing its output at some time step.

model. Simulink's *block reduction optimization* feature statically identifies and removes all dead blocks from a given model [20]. We have not yet explored other possibilities to generate EMI models (e.g, adding blocks in dead execution path or stochastically removing some dead blocks instead of removing all dead blocks) and leave the task as future work.

In our experiments, we noted that CyFuzz-generated models are less likely to contain dead blocks. CyFuzz used to connect all output ports from a model to certain blocks which guaranteed their participation during simulation, or used Simulink's *Signal Logging* feature to record every block's outputs. As a result, most of the blocks[10] participated in simulation and code generation of the model. To enable EMI- based testing, RandGen leaves output ports of randomly chosen blocks unconnected which essentially makes the entire data propagation path involving them dead. We also use blocks to introduce conditional execution of `subsystems` which can be used by EMI generators capable of identifying dead execution paths (e.g., by dynamically collecting coverage information from the model).

### C. Improvements in random generation phases

To support expressive random model creation and using them to find bugs in the Simulink tool chain, we present following improvements over CyFuzz:

*1) Select Blocks Phase:* RandGen supports various new build-in libraries including `Logic and Bit Operations`, `Math Operations` and `Ports and Subsystems` libraries. CuFuzz had basic support for hierarchical model creation whereas RandGen can generate fairly large models with configurable hierarchy depth. When creating child models, RandGen now checks if the child model is compilable, before including it in a parent model. It also reuses some of its previously generated models in hierarchical model creation since creating child models is a time consuming operation. This phase also initializes the internal graph data-structure which represent the random model to be created.

*2) Pre-connection Phase:* This phase ensures that various requirements enforced by different blocks are fulfilled before establishing data-flow relationship between them. As a concrete example, `If` blocks entails presence of special `If Action Subsystems`, one for each of the conditional branches the `If` block induces. To support a wide variety of rules required by specific blocks, RandGen maintains an extensible registry of block-specific *callback functions*. For example, certain callback functions are executed when RandGen sees an `S-Function` block, which call external Csmith utility to generate a random body for the `S-Function` block.

*3) Connect Ports Phase:* Whereas CyFuzz randomly chose unconnected ports, connected them and relied on later phases to recover from illegitimate connections, RandGen build many of the connections the right way. For example, an output port of a `If` block can only be connected to a special `Action` port of some other subsystem. Besides, there can be no other blocks in this data-propagation path. RandGen fulfills such requirements along with updating the model-representing graph data-structure.

*4) Analyze Model Phase:* RandGen analyzes the generated model in this phase using traditional graph search algorithms and utilizing the specification database. One example utility is *algebraic loop*[11] elimination. CyFuzz heavily used Simulink's `getAlgebraicLoops` API in the Fix Errors phase to collect all algebraic loops. However, after identifying several issues with the API, RandGen stopped relying on it and now removes such loops in this phase. Using a depth-fast search traversal on the graph representation of the model, RandGen identifies *back-edges* and places special blocks (e.g. `Delay` block) between the blocks with the back-edges, which breaks the algebraic loop. While this is straight-forward in most cases, care has to be taken for certain edges, e.g., no other blocks can be placed between the connection of `If` block and a subsystem which it drives.

Another analysis RandGen performs is data-type propagation, to reduce data-type mismatch error related fixings in the Fix Errors phase. RandGen sorts the blocks at the beginning of the phase; blocks which data-type is already known (e.g. `Source` blocks and non-direct feed through blocks) are placed at the beginning of the sorted list of blocks, with no sorting order between them. Data-type information is then propagated to other blocks which takes output of these blocks as inputs. To fix data-type mismatches, RandGen place `Data-type conversion` blocks automatically in this phase, in contrast to CyFuzz which compiled a model repeatedly in the Fix Errors phase to identify such data-type inconsistencies. However, RandGen can only support scalar data-types currently.

*5) Fix Errors Phase:* Although we use specifications to generate legitimate models in the first place, due to our incomplete collection of specifications, RandGen still utilizes this phase to some extent. One new addition is that previously, CyFuzz did not fix child models before placing them in the parents, resulting in plentiful of erroneous models which would not compile. To address this issue RandGen fixes errors from child models and uses only those which compile. However, even after using all-legitimate models as descendants, a parent model might not compile since certain data-flow requirements only become apparent when compiling the entire model hierarchy as a whole. RandGen maintains a tree of *generators*[12] and determines the appropriate generator to use to fix models.

## V. Evaluation

In this section we evaluate our contributions based on empirical studies and experiment results.

[10] Some blocks (e.g. `Gain` blocks which multiply inputs by 1) will still be eliminated by Simulink block reduction

[11] Algebraic loops are undesired feedback loops in the model, see [20]

[12] A generator is the object RandGen uses to generate one specific random model. Each top model and child models have their own generator objects.

### A. Research questions and experiment setup

Throughout this work, we explore following broad research questions:

- RQ1 What insight does the study on public Simulink models provide?
- RQ0 Can RandGen effectively fuzz-test Simulink to find bugs in the popular development tool chain?

To answer RQ0, we ran RandGen on five virtual machines, each with 4 gigabytes of RAM and 4 processor cores. We primarily used Matlab/Simulink version R2015a with Ubuntu 16.04 64-bit operating system. Within a four-month long experiment period, RandGen continuously generated random Simulink models and used them to fuzz-test Simulink. Please note that not all of the features described in Section IV-C were available in all of the generated models, as we incrementally added new features and deployed them to our test-setup.

### B. A study on public Simulink models (RQ1)

We discuss our observation on the collected metrics (see Section III-B) in this section. We also investigate how different are the models collected from the four sources.

*1) Model size and organization:* From Table **??**, we note that the median values for total number of blocks for all four classes are fairly small (less than 105); we observe similar pattern for total number of connections in the model. We also note that most of the models are organized in hierarchy (e.g. 41 of the total 42 models from Examples class have hierarchy). Median value of the maximum hierarchy depth does not extend 4 for any of the classes.

While many of the models are small in size, we see some fairly large models in terms of total number of blocks and connections. The largest value for the number of blocks (15,029), number of connections (6343) and maximum hierarchy depth(13) all comes from the Matlab Central class.

We were also interested in distribution of the blocks in different hierarchy levels of a model. For most of the models in all four model classes, number of blocks across hierarchy levels does not vary abruptly. Furthermore, the number of blocks in each hierarchy level is small (less than 40) for most of the models. However, we observe an interesting pattern for number of connections, as they seem to increase as we explore child models in deeper hierarchy levels (Fig. 7).

*2) Observation on model sources:* Here we distinguish model sources based on the metrics obtained from the models in the sources. For example, we readily see that models from the Matlab Central class are more complex than models from any other sources, as determined by the MathWorks cyclomatic measure (Fig. 10). We also note that most of the Examples models are similar to Matlab Central models in terms of size (number of blocks and connections), however, they are less complex than Matlab Central models. In our experiments, most of the GitHub models are "smaller" in size and complexity compared to other classes, however, we see significant number of outliers (Fig. 3, Fig. 10) in this class. This indicates that some of the models in this class are fairly large and complex when compared to other classes.

The distribution of number of blocks per hierarchy level is similar in all four model classes (Fig. 6).

*3) Participation of Simulink libraries in model construction:* Another insight we wanted to explore is from which of the built-in libraries many of the blocks come from. Are most of the blocks in the models built-in blocks or are they mostly custom blocks? Data in Fig. 11 suggests that most of the blocks are built-in blocks, whereas a small portion of the blocks are custom blocks (denoted by the label *User_Defin*). For all four classes, `Ports & Subsystems` and `Math Operations` are the two heavily used libraries.

### C. RandGen found new bugs in Simulink (RQ0)

To date, we have found and reported 8 issues in the Simulink development tool chain. Once reported using MathWork's bug reporting[13] website, the technical support team evaluates it communicating with developers if required and assigns a *Technical Support Case (TSC)* number. Eventually, they might identify the case as bug or enhancement. Table II shows a summary of all the reported bugs. MathWorks confirmed 7 of our reported issues as unique bugs. For one issue they did not identify the case as bug but as a limitation of the block reported and the developers may address its severity once triaged.

In the following text, We discuss some of the bugs in more details. We hand-reduced some of the models and present them as illustrative examples to explain the bugs; these models are available for download[14].

*1) Bugs discovered by EMI-testing:* As discussed in Section IV-B, EMI testing generates an equivalent (modulo input) model from an original model. We used Simulink's block reduction optimization feature to generate EMI variants from a random model and noticed that the feature works incorrectly in Accelerator mode, yielding a difference in behavior in Normal mode simulation. After reporting, the issue was identified as bug (TSC-02476742). In Fig. 13, we present a child model (along with its parent model in the bottom-right corner of the figure) which illustrates the bug. In Normal mode, Simulink reduced all the blocks of the child model except blocks *bl11* and *bl12*, since the reduced blocks are not connected to any output block, hence essentially being *dead*. However, Simulink failed to reduce the blocks in the Accelerator mode, which is a bug.

*2) Bugs discovered by hierarchical model generation:* As we started to generate large hierarchical models, we started noticing Matlab's `getAlgebraicLoops` API hanging and making the entire tool chain unresponsive (TSC-02513701). Other bug (TSC-02472993) manifests when Simulink fails to handle two blocks which are operating at different sample times, leading to a compilation failure. The bug only occurs for a specific block (`First-Order Hold`) and when the block is inside some child model. We present a reduced example in Fig. 14. Next, in Fig. 15 we show a reduced model for bug TSC-02515280. Here, Simulink wrongly counts the number of `Subsystems` for the *Top Model::Subsystem* block.

[13] Available: https://www.mathworks.com/support/bugreports

[14] Available: TODO

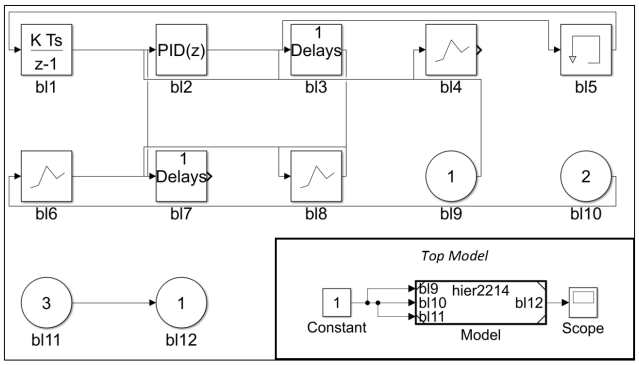


Fig. 13: Simulink model to reproduce TSC-02476742. Top model is shown in the bottom-right corner, its *Model* block refers to the outer model as a child model. In the child model, blocks *bl9*, *bl10* and *bl11* accepts input from the top model and sends output back to the parent using *bl10* block.

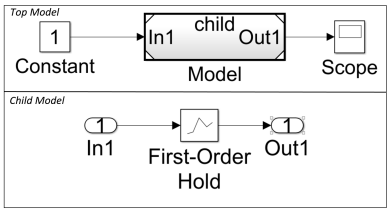


Fig. 14: Simulink model to reproduce bug TSC-02472993. Top model's *Model* block refers to the child model shown in the bottom-half of the figure. Two blocks in this example operate at different sampling rates, and Simulink fails to manage rate transition automatically leading to a compilation failure, which is a bug.

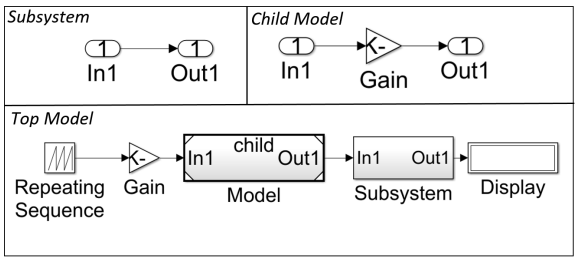


Fig. 15: Simulink model to reproduce bug TSC-02515280. Simulink's *Verification and Validation* toolbox calculates wrong value for the `SubSystemCount` metric for the top model.

*3) Specification mismatch related bugs:* After incorporating Simulink specifications in various phases of RandGen, we identified blocks not implementing specifications correctly. For example, Simulink s `PID Controller (2DOF)` block from Simulink library accepts data of type unsigned integers, however, according to the specification the block only accepts data of type double. (TSC-02386732). While these cases are confirmed as bug, another issue (TSC-02382873) where Simulink block `Sum` fails to properly detect output data type using *internal rule*, is not classified as a bug by MathWorks. However, in our evaluation we posit the behavior as a strong limitation of the block.

## VI. Discussions

### A. Limitations and threats to validity

In this paper, we first perform an empirical study on publicly available Simulink models to collect and observe various properties from them, which can be utilized to generate random real-world models. While some of the models we collected from various sources are large and complex, we note that many of them may be "toy" examples. For example, most of the models from GitHub class are less complex than those from the Examples class.The models we collected may not represent the ones used in the industry. Unfortunately, access to industry-grade CPS models are restricted and such unavailability can thwart the objectives of our project.

## VII. Related Works

Differential testing is common technique that finds applications in compiler fuzzers. Broadly classified, these can be of two types. Compilers for common programming languages can be verified using semantic fuzzers. The most well known is CSmith[5] which verifies any given C compiler against the C99 standard. Data-flow languages, however, are tested using data-flow fuzzing, as used by CyFuzz[1]. Input generation and comparison for both these work significantly differently.

Our work heavily draws on Vu Le et al. and their analysis of Equi-Modulo Inputs to verify different compilers[6]. Also related is Nyugen et al.[22] and Csallner et al.[23] and their work on a differential testing based runtime verification framework, as well as Sampath et al.'s approach in testing model-processing tools by using Stateflow(a Simulink component)'s semantic meta-model[24]

Various empirical studies computes useful properties from open source Java program [16], [15], [14]. Random Java program generator RUGRAT collects its various configuration parameters from empirical studies on Java source and bytecode [25].

## VIII. Conclusion

Our work focuses on enhancing Simulink model generators by studying real-world models from different sources. We've been able to compute certain specific metrics from these sources that have gone into improving Cyfuzz's random model generation algorithm. Thus we can meaningfully claim that the models generated by Cyfuzz are semantically close and structurally similar to the so-called average of all the models that are available for study in public domains.

Also our study of EMI techniques has gone into improving the range of possibilities of models that we are able to generate. This strengthens the existing fuzz-testing approach and helps diversify the kind of bugs detected by Cyfuzz.

Both these approaches show utility in the fact that Cyfuzz was able to correctly detect 7 bugs that were verified and approved as valid bugs by the Simulink development and support team. Thus the merit of our approaches stands verified.

TABLE II: Simulink bugs RandGen discovered and reported. Headers *TSC*, *St* and *Ver* denotes Technical Support Case number for the bug, status (whether the bug is confirmed) and latest version of Simulink where we could reproduce the bug.

| TSC | Summary | St | Ver |
|---|---|---|---|
| 02513701 | `getAlgebraicLoops` hangs for large models with hierarchy and makes Simulink unresponsive | ✓ | 2015a |
| 02476742 | `Acceleration` mode simulation with block-reduction optimization does not work | ✓ | 2017a |
| 02472993 | *Automated rate transition* does not work in hierarchical models for specific block | ✓ | 2017a |
| 02515280 | Verification and Validation toolbox computes wrong value for `SubSystemCount` metric | ✓ | 2017a |
| 02386732 | Data-type support specification for `PID Controller (2DOF)` block does not match specification | ✓ | 2015a |
| 02382544 | Simulink block `Constant` does not have a parameter as specified in the documentation | ✓ | 2015a |
| 02382873 | Internal rule can not identify data-type for `Add` block | X | 2015a |

## REFERENCES


[1] S. A. Chowdhury, T. T. Johnson, and C. Csallner, "Cyfuzz: A differential testing framework for cyber-physical systems development environments," in *Proc. 6th Workshop on Design, Modeling and Evaluation of Cyber Physical Systems (CyPhy)*. Springer International Publishing, Oct. 2016.

[2] C. Guger, A. Schlogl, C. Neuper, D. Walterspacher, T. Strein, and G. Pfurtscheller, "Rapid prototyping of an eeg-based brain-computer interface (bci)," *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, vol. 9, no. 1, pp. 49–58, March 2001.

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

[4] K. Dewey, J. Roesch, and B. Hardekopf, "Fuzzing the Rust typechecker using CLP (T)," in *Proc. 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 482–493. [Online]. Available: http://dx.doi.org/10.1109/ASE.2015.65

[5] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, Jun. 2011, pp. 283–294. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993532

[6] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, 2014, pp. 216–226. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594334

[7] J. Ruderman, "Introducing jsfunfuzz," https://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/, 2007.

[8] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proc. 21th USENIX Security Symposium*. USENIX Association, Aug. 2012, pp. 445–458. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[9] I. Stürmer, M. Conrad, H. Dörr, and P. Pepper, "Systematic testing of model-based code generators," *IEEE Transactions on Software Engineering (TSE)*, vol. 33, no. 9, pp. 622–634, Sep. 2007. [Online]. Available: http://dx.doi.org/10.1109/TSE.2007.70708

[10] G. Hamon and J. Rushby, "An operational semantics for Stateflow," *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 5, pp. 447–456, 2007. [Online]. Available: http://dx.doi.org/10.1007/s10009-007-0049-7

[11] O. Bouissou and A. Chapoutot, "An operational semantics for Simulink's simulation engine," in *Proc. 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES)*. ACM, Jun. 2012, pp. 129–138. [Online]. Available: http://doi.acm.org/10.1145/2248418.2248437

[12] The MathWorks Inc., "Products and services," http://www.mathworks.com/products/, 2016.

[13] A. C. Rajeev, P. Sampath, K. C. Shashidhar, and S. Ramesh, "CoGenTe: A tool for code generator testing," in *Proc. 25th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, Sep. 2010, pp. 349–350. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859070

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

[15] C. S. Collberg, G. Myles, and M. Stepp, "An empirical study of java bytecode programs," *Softw., Pract. Exper.*, vol. 37, no. 6, pp. 581–641, 2007. [Online]. Available: http://dx.doi.org/10.1002/spe.776

[16] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi, "An empirical investigation into a large-scale java open source code repository," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '10. New York, NY, USA: ACM, 2010, pp. 11:1–11:10. [Online]. Available: http://doi.acm.org/10.1145/1852786.1852801

[17] K. Sinha and O. de Weck, "Structural complexity metric for engineered complex systems and its application," in *14TH INTERNATIONAL DEPENDENCY AND STRUCTURE MODELLING CONFERENCE, DSM12*, 2012.

[18] M. Olszewska, Y. Dajsuren, H. Altinger, A. Serebrenik, M. A. Waldén, and M. G. J. van den Brand, "Tailoring complexity metrics for simulink models," in *Proccedings of the 10th European Conference on Software Architecture Workshops, Copenhagen, Denmark, November 28 - December 2, 2016*, 2016, p. 5. [Online]. Available: http://dl.acm.org/citation.cfm?id=3004853

[19] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann, "SimCoTest: A test suite generation tool for Simulink/Stateflow controllers," in *Proc. 38th International Conference on Software Engineering, (ICSE)*. ACM, May 2016, pp. 585–588. [Online]. Available: http://doi.acm.org/10.1145/2889160.2889162

[20] The MathWorks Inc., "Simulink documentation," http://www.mathworks.com/help/simulink/, 2017, accessed May 2017.

[21] ——, "Types of Model Coverage - Matlab & Simulink," https://www.mathworks.com/help/slvnv/ug/types-of-model-coverage.html, 2017, accessed May 2017.

[22] L. V. Nguyen, C. Schilling, S. Bogomolov, and T. T. Johnson, "HyRG: A random generation tool for affine hybrid automata," in *Proc. 18th International Conference on Hybrid Systems: Computation and Control (HSCC)*. ACM, Apr. 2015, pp. 289–290. [Online]. Available: http://doi.acm.org/10.1145/2728606.2728650

[23] C. Csallner and Y. Smaragdakis, "JCrasher: An automatic robustness tester for Java," *Software—Practice & Experience*, vol. 34, no. 11, pp. 1025–1050, Sep. 2004. [Online]. Available: http://dx.doi.org/10.1002/spe.602

[24] P. Sampath, A. C. Rajeev, S. Ramesh, and K. C. Shashidhar, "Testing model-processing tools for embedded systems," in *Proc. 13th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, Apr. 2007, pp. 203–214. [Online]. Available: http://dx.doi.org/10.1109/RTAS.2007.39

[25] I. Hussain, C. Csallner, M. Grechanik, Q. Xie, S. Park, K. Taneja, and B. M. Hossain, "Rugrat: Evaluating program analysis and testing tools and compilers with large generated random benchmark applications," *Software—Practice & Experience*, vol. 46, no. 3, pp. 405–431, Mar. 2016.