# CyFuzz: A Differential Testing Framework for Cyber-Physical Systems Development Environments

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

The University of Texas at Arlington, Texas, USA
shafiulazam.chowdhury@mavs.uta.edu
{taylor.johnson,csallner}@uta.edu

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

**Keywords:** Differential testing, cyber-physical systems, model-based design, Simulink

## 1 Introduction

Widely used cyber-physical system (CPS) development tool chains are complex software systems that typically consist of millions of lines of code [1]. For example, the popular MathWorks Simulink tool chain contains model-based design tools (in which *models* in various expressive modeling languages are used to describe the overall system under control [2]), simulators, compilers, and automated code generators. Like any complex piece of code, CPS tool chains may contain bugs and such bugs may lead to severe CPS defects.

The vast majority of resources in the CPS design and development phases are devoted to ensure that systems meet their specifications [3, 4]. In spite of having sophisticated design validation and verification approaches (model checking, automated test case generation, hardware-in-the-loop and software-in-the-loop testing etc.), we see frequent safety recalls of products and systems among industries, due to CPS bugs [5–7].

Since many CPSs operate in safety-critical environments and have strict correctness and reliability requirements [8], it would be ideal for CPS development tools to not have bugs or unintended behaviors. However, this is not generally true as demonstrated by recent *random testing* projects finding bugs in a static analysis tool (Frama-C) [9] and in popular C compilers (GCC and LLVM) [10], which are widely used in CPS model-based design.

It would be extremely expensive or possibly even practically infeasible to formally verify entire CPS tool chains. In addition to their sheer size in terms of lines of code, a maybe more significant hurdle is the lack of a complete and up to date formal specification of the CPS tool chain semantics, which may be due to their complexity and rapid release cycles [1, 11].

Instead of formally verifying the absence of bugs in all CPS tool chain execution paths, we revert to showing the presence of bugs on individual paths (aka testing), which can still be a major contributor to software quality [12]. *Differential testing* or *fuzzing*, a form of random testing, mechanically generates random test inputs and presents them to comparable variations of a software [12]. The results are then compared and any variation from the majority (if one exists) likely indicates a bug [13]. This scheme has been effective at finding bugs in compilers and interpreters of traditional programming languages. As an example, various fuzzing schemes have collectively found over 1,000 bugs in widely used compilation tools such as GCC [10, 11, 14].

While compiler testing is promising, when testing CPS tool chains we face additional challenges beyond what is covered by testing compilers of traditional programming languages (such as Csmith creating C programs), since CPS modeling languages differ significantly from traditional programming languages. A key difference is that the complete semantics of widely used commercial modeling languages (e.g., MathWorks Simulink and Stateflow [15]) are not publicly available [1, 16, 17]. Moreover, modeling language semantics often depend on subtle details, such as two-dimensional layout information, internal model component settings, and the particular interpretation algorithm of simulators [1]. Finally, random generation of test cases for CPS development environments has to address a combination of programming paradigms (e.g., both graphical, data-flow language and textual imperative programming language in the same model), which is rare in traditional compiler testing.

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

We also present an implementation for testing the Simulink environment, which is widely used in CPS industries for model-based design of dynamic and embedded systems [18, 19]. Although our current prototype implementation targets Simulink, the described conceptual framework is not tool specific and should thus be applicable to related CPS tool chains, such as NI's LabVIEW [20].

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

- To understand the types of Simulink bugs that affect users, we first analyze a subset of the publicly available Simulink bug reports (Section 3).
- We present CyFuzz, a conceptual framework for (1) generating random but valid models for a CPS modeling language, (2) simulating the generated models on alternative CPS tool chain configurations, and (3) comparing the simulation results (Section 4). We then describe interesting implementation details and challenges of our prototype implementation for Simulink (Section 5).
- We report on our experience of running our prototype tool on various Simulink configurations (Section 6), identifying comparison errors and semi-independently reproducing a confirmed bug in Simulink's `Rapid Accelerator` mode.

## 2 Background: Model-based CPS Design and Simulink

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

### 2.1 CPS Model Elements

The following concepts and terms are applicable to many CPS modeling languages (including Simulink). A *model*, also known as a *block-diagram*, is a mathematical representation of some CPS [18]. Designing a diagram starts with choosing elementary elements called blocks. Each *block* represents a component of the CPS and may have *input* and *output* ports. An input port accepts data on which the block performs some operation. An output port passes data to other input ports using *connections*. An output port can be connected to more than one input port while the opposite is not true in general. A Block may have *parameters*, which are configurable values that influence the block's behavior. Somewhat similar to a programming language's standard libraries, a CPS tool chain typically provides *block libraries*, where each library consists of a set of predefined blocks.

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

When simulating, the SUT numerically solves the mathematical formulas represented by the model [18]. Simulation is usually time bound and at each *step* of the simulation, a *solver* calculates the blocks' outputs. We use the term *signal* to mean output of a block's port at a particular simulation step.

The very first phase of the simulation process is *compiling* the model. This stage also looks for incorrectly generated models and raises failures for syntactical model errors, such as data type mismatches between connected output and input ports. If an error is found in the compilation phase, the SUT does not attempt simulating the model. After successful simulation, *code generators* can generate native code, which may be deployed in target hardware [1].

### 2.2 Example CPS Development Environment: Simulink

While our conceptual framework uses the above terms, they also apply directly in the context of Simulink [21]. Besides having a wide selection of built-in blocks, Simulink allows integrating native code (e.g., Matlab or C code) in a model via Simulink's `S-function` interface, which lets users create custom blocks for use in their models. Simulink's `Subsystem` and `Model referencing` features enable hierarchical models.

Simulink has three simulation modes. In `Normal` mode, Simulink does not generate code for blocks, whereas it generates native code for certain blocks in the `Accelerator` mode. Unlike in these two modes, the `Rapid Accelerator` mode further creates for the model a standalone executable. To capture simulation results we use Simulink's `Signal Logging` functionality as we found implementing it quite feasible. However, for cases where the approach is not applicable (see [21]), we use Simulink's `sim` api to record simulation data.

## 3 Study of Existing Bugs: Incorrect Code Generation

To understand the types of bugs Simulink users have found and care about, we performed a study on the publicly available bug reports from the MathWorks website[1]. We identified commonalities in bug reports, which we call *classification factors*. We limited our study to bug reports found via the search query *incorrect code generation*, as earlier studies have identified code generation as vulnerable [1, 22].

We investigated bug reports affecting Matlab/Simulink version 2015a as we were using it in our experiments. As of February 17, 2016, there were 50 such bug reports, among which 47 have been fixed in subsequent releases of the products. Table 1 summarizes the findings. Our complete study data are available at: `http://bit.ly/simstudy`

Table 1 shows only those classification factors that affect at least 20% of all the bug reports that we have studied. We use insights obtained from the

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

**Table 1.** Study of publicly available Simulink bug reports. The right column denotes the percentage of bug reports affected by a the given classification factor. Each bug report may be classified under multiple factors.

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

study in our CyFuzz prototype implementation. For example, many of the bug reports (54%) are related to simulation result and generated code execution output mismatch. Thus, differential testing (e.g., by comparing simulation and code execution) seems like a good fit for finding bugs in CPS tool chains. Further insight that is reflected in our tool is that it is worth exploring the large space of possible block connections (36% of bug reports) e.g., via random block and connection generation. Other insights we want to use in the future are to incorporate random block parameter values and port data-types (56%) and model configurations (32%).

## 4 Differential Testing of CPS Development Tool Chains

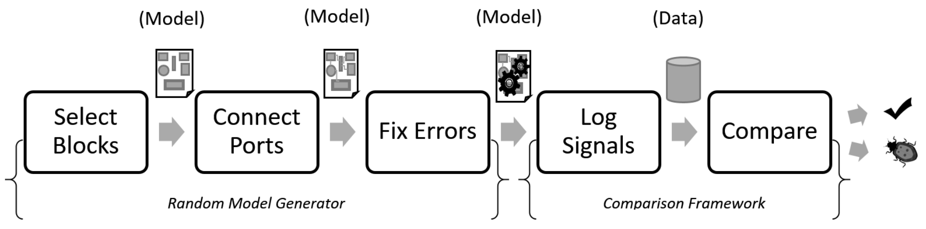

**Fig. 1.** Overview of the differential testing framework. The first three *phases* correspond to the random model generator, while the rest belongs to the comparison framework.

At a high level we can break our objective into two sub goals: creating a *random model generator* and defining a *comparison framework*. We first present a theory applicable to a conceptual CPS framework in this section. Fig. 1 provides a schematic overview of CyFuzz's processing *phases*. The first three phases belong to the random model generator, and the remaining two constitute the comparison framework. The first two phases create a random model (which may violate Simulink's model construction rules). The third phase fixes many of these errors, such that the model passes the SUT's type checkers and the SUT can simulate it. If it succeeds it passes the model to the fourth phase to simulate the model in various SUT configurations and to record results. The final phase detects any dissimilarities in the collected data, which we call *comparison error* bugs.

## 4.1 Conceptual Random Model Generator

Following are details on the generator's three phases.

**Listing 1.1.** Select Blocks phase of the conceptual random model generator.

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

**Select Blocks.** Listing 1.1 summarizes this phase, which selects, places, and configures the model's blocks. The generator has a list of block libraries and for each library a predetermined weight. Using the weights, the *choose_blocks* method selects $n$ random blocks. The value $n$ can be fixed or randomly selected from a range. On a newly created model the generator next places each of these blocks using the *place_block_in_model* method. For creating inputs, CyFuzz selects various kinds of blocks, to, for example, provide random inputs to the model.

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

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

**Listing 1.2.** $fix\_errors$ tries to fix the model errors that the *simulate* method raises; $p$ is a SUT configuration; $t$ denotes a timeout value.

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

**Fix Errors.** Because of their simplicity, CyFuzz's first two phases may generate invalid models that cannot be simulated successfully. The third phase tries to fix these errors. Listing 1.2 outlines the approach. It uses method *simulate* to simulate model $m$ up to time $t \in \mathbb{R}^+$ (in milliseconds) using SUT configuration $p$.

The *simulate* output is a 3-tuple, where $r^p_{status}$ is one of $success$, $error$, or $timed - out$. Note that first step of simulation is compiling the model (see Section 2). If $m$ has errors, *simulate* will abort compilation, storing error-related diagnostic information in $errors$. $r^p_{data}$ contains simulation results (time series data of the model's blocks' outputs) if $r^p_{status} = success$.

At this point we assume that the error messages are informative enough to drive the generator. For example, Simulink satisfies this assumption. Using $errors$, $fix\_model$ tries to fix the errors by changing the model. As it changes the model this phase may introduce new errors. We try to address such secondary errors in subsequent loop iterations in Listing 1.2, up to a configurable number *attempt_limit*. While this approach is clearly an imperfect heuristic, it has worked relatively well in our preliminary experience (as, e.g., is indicated by the low error rate in Table 2).

## 4.2 Conceptual Comparison Framework

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

**Log Signals.** If simulation was successful in the Fix Errors phase, CyFuzz simulates the model varying configurations of the SUT in this phase; let $P$ be such a set of configurations. Using the *simulate* method introduced in Section 4.1, for each $p \in P$ we calculate $< r^p_{status}, r^p_{data}, errors >= simulate(m, p, t)$ for a model $m$ and add $r^p_{data}$ to a set $d$ only if $r^p_{status} = success$. We pass $d$ to next phase of the framework. $r^p_{data}$ should contain time series data of the output ports of the model's blocks at all available simulation steps. In the next phase, however, we use only the values recorded at the last simulation step; we leave comparing signal values at other simulation steps as future task.

**Compare.** In its last phase, CyFuzz compares the recorded simulation results $d$ obtained in the previous phase using method *compare* (Listing 1.4). It uses method *retrieve*, which returns the signal value of a particular block's particular port at a given time instance. If the value is not available (e.g., blocks that do not have output ports do not participate in signal logging), it returns the special value *Nil*. *compare* also uses method *latest_time* which returns the time of the last simulation step for a given block's particular port. If no data is available, it returns *Nil*.

**Listing 1.3.** Determining equivalence via tolerance limit $\epsilon$.

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

**Listing 1.4.** This method compares two execution results (of model $m$) taken as first two arguments and throws errors if it finds a dissimilarity.

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

Now, taking two elements from $d$ at a time we form all possible pairs $(r^p_{data}, r^q_{data})$ where $p \neq q$ and apply method *compare* on them. As comparing floating-point numbers using straight equality checking is problematic [1, 23], *eqiv* (Listing 1.3) method uses a tolerance limit to determine floating-point equivalence. If *compare* reports an error, we mark $m$ as a *comparison error* for $p, q$ and submit it to manual inspection.

## 5 CyFuzz Prototype Implementation for Simulink

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

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

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

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

*Models with Random Native Code.* To facilitate blocks with custom behavior, Simulink allows placing native code (C, Matlab etc.) directly in models. To generate such blocks we leverage Csmith, which generates random C programs [10]. We designed simple Simulink blocks using Matlab's `S-function` interface that use random code generated by a customized version of Csmith. Our customized version is capable of generating many different C functions that can be called from various simulation steps. We looked for both crash errors and "wrong code errors" (similar to our comparison error). However, this is not fully integrated with CyFuzz yet.

*The Comparison Framework.* CyFuzz starts with varying simulation modes (see Section 2.2). and compiler optimization levels. For instance, "`Normal` mode", "`Accelerator` mode; optimization on", and "`Rapid Accelerator`; optimization off" are options to vary. Varying compilers, code generators, solver-specific settings, and other possible SUT configuration options are future work.

## 6 Experience with CyFuzz

Here we analyze our prototype implementation based on experimental results.

### 6.1 Research Questions (RQ), Experimental Setup, and Results

Throughout this work we explore the following research questions.

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

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

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

To answer these questions we conducted experiments using Matlab `2015a` on Ubuntu 14.10 and varied simulation mode (`Normal` vs. `Accelerator`) and optimizer (on vs. off) for the later mode. For the *fix_errors* method (Listing 1.2) we chose *attempt_limit* 10 and *timeout* 12. For choosing blocks we used a traditional $O(n)$ implementation of the *fitness proportion selection* algorithm [24]. We have not included in these experiments hierarchical models or custom blocks.

**Table 2.** Each row represents a separate experiment. Columns 3–6 is the percentage of blocks selected per library (e.g., experiment A chose 80% of the blocks from the `Discrete` library). *Error* denotes the number of models that failed to simulate. *Timed-out* denotes the models that did not complete simulation within the time bound.

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

**Table 3.** More information on experiments from Table 2. Columns 3-7 denotes the time taken by the five phases of CyFuzz. *Runtime* denotes the average time CyFuzz spent for a model.

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

*Effectively Creating Random Models (RQ 1).* As the experimental results in Table 2 suggest, our tool can generate many models that Simulink can successfully simulate. For each row in the table we have a low error and timed-out rate. This high success rate is crucial for the framework as it only uses such valid models in the tool's later comparison framework phases. We also observed that the number of errors and timed-out models varied with the selected block libraries, but we have not yet analyzed the reasons of these variations.

*Effectiveness of Comparison Framework (RQ 2).* We have not found new bugs yet, however, our framework reproduced an existing bug and found interesting cases (see Section 6.2).

*Runtime Analysis (RQ 3).* The Select Blocks algorithm of Listing 1.1 has runtime $O(n)$, $n$ being the number of blocks in the model and using an $O(1)$ block selection algorithm. The random model generator scales linearly with the number of blocks. But as the number of blocks grows, the number of timed-out models and errors also grow. A preliminary analysis suggests that there are relatively few distinct error causes. We group errors by their causes and fixing one cause dramatically increased the overall number of successfully executed models.

Table 3 indicates that the Log Signals phase uses most of the runtime. This result is not surprising, as in this phase the SUT simulates the model, generates and executes code, and logs the data, all of which are time consuming tasks.

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

### 6.2 Interesting Comparison Framework Findings

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

**Comparison Error for Models with Algebraic Loops.** In our experiments we noticed comparison errors for some models where Simulink solved algebraic loops. Investigating further we noticed that when Simulink solves an algebraic loop it is not confident of its correctness [21]. For this, we did not classify this case as a bug. CyFuzz now eliminates algebraic loops altogether rather than relying on Simulink to solve them. We note that one can use our tool to opportunistically discover such inaccuracies for models with algebraic loops and decide whether to accept Simulink's solution for solving the loops.

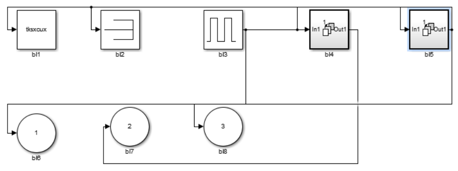

**Fig. 2.** Screen-shot of generated top-level Simulink model which reproduced a bug

**Bug in Simulink's `Rapid Accelerator` Mode.** In separate experiments with hierarchical models, we noticed that for a model (see Fig. 2) values of a Simulink `Outport` block are significantly different in `Normal` and `Rapid Accelerator` mode. This was detected automatically by our comparison framework. After submitting a bug report MathWorks confirmed that the case was already identified as a bug and they fixed it for later versions.

## 7 Future Work and Discussion

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

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

The comparison framework implementation is also not free from shortcomings. So far, we have only used various simulation modes and compiler optimization levels. However, we are interested in adding more variations (e.g. those listed in Section 5). Finally, CyFuzz should compare signals in multiple simulation steps, since it was also found effective in previous work [25].

## 8 Related Work

The following focuses on the most closely related work not covered by the introduction section. Existing approaches for CPS testing mostly aim at generating test cases for existing models (e.g., [26, 18]) and do not target testing of CPS tool chains. *Code generator testing* ([1, 27]) only target a relatively small component of the CPS tool chain but not an entire CPS tool chain.

Most of the compiler fuzzers perform random walks over a context-free grammar, thus mainly focusing on generating syntactically valid [14] and well typed programs in imperative languages [28, 10, 11, 29]. None of the works target dataflow languages like Simulink. We find Csmith most related to our work, which is state-of-the-art C compiler fuzzer. Csmith leverages the well-published C99 standard and can be used to test only a component of entire CPS tool chain [10]. Our test generation and comparison techniques differ fundamentally from Csmith. Conceptually, CPS tool chain fuzzing is a super-set of the schemes presented in Csmith. CPS tool chains typically contain a C compiler; thus CyFuzz leverages Csmith as a component.

Earlier work includes a differential testing based runtime verification framework, leveraging a random hybrid automata generator [30, 25]. Other works attack code generators used in CPS tool chain. Stürmer et al. generate model taking specification of a code generator's optimization rules in graph grammar [1]. But such specifications for code generators might not be available and white-box testing in parts is undesirable [31]. Sampath et al. propose testing *model-processing tools* taking semantic meta-model of Stateflow (a Simulink component) [31]. But the approach does not scale and the complete specifications it needs are not available. In contrast, we propose the first fuzz-testing framework to test arbitrary CPS tool chains based on feasible model generation.

Many CPS model verification and safety checking approaches have been proposed [8, 32]. Recent work verifies existing SL/Stateflow (SL/SF) models by generating test inputs for these models [18, 19]. Alur et al. analyze generated symbolic traces of a SL/SF model, and combine simulation and symbolic analysis for improving coverage of given SL/SF models [33]. The *Simulink Code Inspector* compares generated code for a given model based on structural equivalence and traceability [21]. However none of these approaches describe random generation of Simulink models for fuzzing the CPS tool chain.

## 9 Conclusions

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

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

*Acknowledgments.* This material is based upon work supported by the National Science Foundation under Grants No. 1117369, 1464311, and 1527398, by Air Force Office of Scientific Research (AFOSR) contract numbers FA9550-15-1-0258 and FA9550-16-1-0246, and by Air Force Research Lab (AFRL) contract number FA8750-15-1-0105. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of AFRL, AFOSR, or NSF.

## References

1. Stürmer, I., Conrad, M., Dörr, H., Pepper, P.: Systematic testing of model-based code generators. IEEE Transactions on Software Engineering (TSE) **33**(9) (September 2007) 622–634
2. Lee, E.A., Seshia, S.A.: Introduction to Embedded Systems: A Cyber-physical Systems Approach. First edn. `http://LeeSeshia.org` (2011)
3. Beizer, B.: Software testing techniques. Second edn. Van Nostrand Reinhold (June 1990)
4. U.S. National Institute of Standards and Technology (NIST): The economic impacts of inadequate infrastructure for software testing: Planning report 02-3 (May 2002)
5. U.S. Consumer Product Safety Commission (CPSC): Recall 11-702: Fire alarm control panels recalled by fire-lite alarms due to alert failure. http://www.cpsc.gov/en/Recalls/2011/Fire-Alarm-Control-Panels-Recalled-by-Fire-Lite-Alarms-Due-to-Alert-Failure (October 2010)
6. U.S. National Highway Traffic Safety Administration (NHTSA): Defect Information Report 14V-053. `http://www-odi.nhtsa.dot.gov/acms/cs/jaxrs/download/doc/UCM450071/RCDNN-14V053-0945.pdf` (February 2014)
7. Alemzadeh, H., Iyer, R.K., Kalbarczyk, Z., Raman, J.: Analysis of safety-critical computer failures in medical devices. IEEE Security Privacy **11**(4) (July 2013) 14–26
8. Johnson, T.T., Bak, S., Drager, S.: Cyber-physical specification mismatch identification with dynamic analysis. In: Proc. ACM/IEEE Sixth International Conference on Cyber-Physical Systems (ICCPS), ACM (April 2015) 208–217
9. Cuoq, P., Monate, B., Pacalet, A., Prevosto, V., Regehr, J., Yakobowski, B., Yang, X.: Testing static analyzers with randomly generated programs. In: Proc. 4th NASA Formal Methods Symposium (NFM), Springer (April 2012) 120–125
10. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), ACM (June 2011) 283–294
11. Dewey, K., Roesch, J., Hardekopf, B.: Fuzzing the Rust typechecker using CLP (T). In: Proc. 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE (2015) 482–493
12. McKeeman, W.M.: Differential testing for software. Digital Technical Journal **10**(1) (1998) 100–107
13. Lidbury, C., Lascu, A., Chong, N., Donaldson, A.F.: Many-core compiler fuzzing. In: Proc. 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), ACM (June 2015) 65–76
14. Holler, C., Herzig, K., Zeller, A.: Fuzzing with code fragments. In: Proc. 21th USENIX Security Symposium, USENIX Association (August 2012) 445–458
15. The MathWorks Inc.: Products and services. `http://www.mathworks.com/products/` (2016)

16. Hamon, G., Rushby, J.: An operational semantics for Stateflow. International Journal on Software Tools for Technology Transfer **9**(5) (2007) 447–456
17. Bouissou, O., Chapoutot, A.: 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 (June 2012) 129–138
18. Matinnejad, R., Nejati, S., Briand, L.C., Bruckmann, T.: SimCoTest: A test suite generation tool for Simulink/Stateflow controllers. In: Proc. 38th International Conference on Software Engineering, (ICSE), ACM (May 2016) 585–588
19. Sridhar, A., Srinivasulu, D., Mohapatra, D.P.: Model-based test-case generation for Simulink/Stateflow using dependency graph approach. In: Proc. 3rd IEEE International Advance Computing Conference (IACC). (February 2013) 1414–1419
20. National Instruments: Labview system design software. `http://www.ni.com/labview/` (2016)
21. The MathWorks Inc.: Simulation documentation. `http://www.mathworks.com/help/simulink/` (2016)
22. Rajeev, A.C., Sampath, P., Shashidhar, K.C., Ramesh, S.: CoGenTe: A tool for code generator testing. In: Proc. 25th IEEE/ACM International Conference on Automated Software Engineering (ASE), ACM (September 2010) 349–350
23. Goldberg, D.: What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv. **23**(1) (1991) 5–48
24. Goldberg, D.E.: Genetic algorithms in search, optimization and machine learning. First edn. Addison-Wesley (1989)
25. Nguyen, L.V., Schilling, C., Bogomolov, S., Johnson, T.T.: Runtime verification of model-based development environments. In: Proc. 15th International Conference on Runtime Verification (RV). (September 2015)
26. Girard, A., Julius, A.A., Pappas, G.J.: Approximate simulation relations for hybrid systems. Discrete Event Dynamic Systems **18**(2) (2008) 163–179
27. Stürmer, I., Conrad, M.: Test suite design for code generation tools. In: Proc. 18th IEEE International Conference on Automated Software Engineering (ASE). (October 2003) 286–290
28. Csallner, C., Smaragdakis, Y.: JCrasher: An automatic robustness tester for Java. Software—Practice & Experience **34**(11) (September 2004) 1025–1050
29. Hussain, I., Csallner, C., Grechanik, M., Xie, Q., Park, S., Taneja, K., Hossain, B.M.: Rugrat: Evaluating program analysis and testing tools and compilers with large generated random benchmark applications. Software—Practice & Experience **46**(3) (March 2016) 405–431
30. Nguyen, L.V., Schilling, C., Bogomolov, S., Johnson, T.T.: HyRG: A random generation tool for affine hybrid automata. In: Proc. 18th International Conference on Hybrid Systems: Computation and Control (HSCC), ACM (April 2015) 289–290
31. Sampath, P., Rajeev, A.C., Ramesh, S., Shashidhar, K.C.: Testing model-processing tools for embedded systems. In: Proc. 13th IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE (April 2007) 203–214
32. Mohaqeqi, M., Mousavi, M.R.: Sound test-suites for cyber-physical systems. In: 10th International Symposium on Theoretical Aspects of Software Engineering (TASE). (July 2016) 42–48
33. Kanade, A., Alur, R., Ivancic, F., Ramesh, S., Sankaranarayanan, S., Shashidhar, K.C.: Generating and analyzing symbolic traces of Simulink/Stateflow models. In: Proc. 21st International Conference on Computer Aided Verification (CAV), Springer (June 2009) 430–445