Detecting Simulink Compiler Bugs via Controllable Zombie Blocks Mutation

Shikai Guo
School of Information Science and Technology, Dalian Maritime University
Dalian, Liaoning, China
shikai.guo@dlmu.edu.cn

He Jiang*
School of Software, Dalian University of Technology
Dalian, Liaoning, China
jianghe@dlut.edu.cn

Zhihao Xu
School of Information Science and Technology, Dalian Maritime University
Dalian, Liaoning, China
cemery@dlmu.edu.cn

Xiaochen Li
Zhilei Ren†
School of Software, Dalian University of Technology
Dalian, Liaoning, China
{xiaochen.li,zren}@dlut.edu.cn

Zhide Zhou
School of Software, Dalian University of Technology
Dalian, Liaoning, China
cszide@gmail.com

Rong Chen
School of Information Science and Technology, Dalian Maritime University
Dalian, Liaoning, China
rchen@dlmu.edu.cn

ABSTRACT

As a popular Cyber-Physical System (CPS) development tool chain, MathWorks Simulink is widely used to prototype CPS models in safety-critical applications, e.g., aerospace and healthcare. It is crucial to ensure the correctness and reliability of Simulink compiler (i.e., the compiler module of Simulink) in practice since all CPS models depend on compilation. However, Simulink compiler testing is challenging due to millions of lines of source code and the lack of the complete formal language specification. Although several methods have been proposed to automatically test Simulink compiler, there still remain two challenges to be tackled, namely the limited variant space and the insufficient mutation diversity. To address these challenges, we propose COMBAT, a new differential testing method for Simulink compiler testing. COMBAT features the combination of an EMI (Equivalence Modulo Input) mutation component and a diverse variant generation component. The EMI mutation component inserts assertion statements (e.g., *If*/*While* blocks) at arbitrary points of the seed CPS model. These statements break each insertion point into true and false branches. Then, COMBAT feeds all the data passed through the insertion point into the true branch to preserve the equivalence of CPS variants. In such a way, the body of the false branch could be viewed as a new variant space, thus addressing the first challenge. The diverse variant generation component uses Markov chain Monte Carlo optimization to sample the seed CPS model and generates complex mutations of long sequences of blocks in the variant space, thus addressing the second challenge. Experiments demonstrate that COMBAT significantly outperforms the state-of-the-art approaches in Simulink compiler testing. Within five months, COMBAT has reported 16 valid bugs for Simulink R2021b, of which 11 bugs have been confirmed as new bugs by MathWorks Support.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Compilers**; *Software reliability*.

KEYWORDS

Cyber-physical system, Simulink, differential testing, compiler bug

ACM Reference Format:
Shikai Guo, He Jiang, Zhihao Xu, Xiaochen Li, Zhilei Ren, Zhide Zhou, and Rong Chen. 2022. Detecting Simulink Compiler Bugs via Controllable Zombie Blocks Mutation. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3540250.3549159

1 INTRODUCTION

A Cyber-Physical System (CPS) is an embedded system in which computational and physical processes are tightly integrated to provide a favorable environment for intelligent information processing, real-time sensing, and physical dynamic control of large-scale engineering systems [27, 28]. As a commercial CPS tool chain, MathWorks Simulink has become an industry standard, which is widely used by engineers to design, model, simulate, and generate embedded code for CPS models. To deploy CPS models in the target hardware of safety-critical applications [20, 22, 35, 50], all CPS models are required to be correctly compiled in Simulink compiler. Therefore, it is crucial to ensure the correctness and reliability of Simulink compiler, because the bugs in Simulink compiler may cause the functionality of developed CPS models

*He Jiang is also with Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province, and DUT Artificial Intelligence Institute, Dalian, China.
†Zhilei Ren is also affiliated with Key Laboratory of Safety-Critical Software (Nanjing University of Aeronautics and Astronautics), Ministry of Industry and Information Technology, Nanjing, China

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3549159

beyond engineers' expectation, which may result in unexpected behavior in safety-critical applications [7, 9].

However, it is difficult to detect bugs in Simulink compiler, since these bugs typically manifest indirectly as CPS model failures. Engineers may wrongly consider these failures as the bugs in CPS models, rather than the bugs caused by Simulink compiler. Additionally, Simulink compiler testing is challenging due to its large and complex codebase [5, 17, 33] and the lack of complete formal language specification [6, 18, 21].

An effective method for testing compilers is Equivalence Modulo Input (EMI). EMI is a differential testing method, which produces equivalent test programs by mutating the unexecuted code in the existing program under a given input [10, 39, 46, 51]. Compiler bugs can be detected by comparing two identically configured compiler executions (on equivalent input programs). EMI-based methods have been proven to be effective for detecting thousands of bugs in C compilers [11, 24–26, 44, 45].

However, existing EMI-based methods cannot be directly applied to testing Simulink compiler because the CPS language is different from traditional programming languages [13, 14]. On the one hand, the notion of determining 'unexecuted code' (which are referred as zombie regions in CPS models) is different [13, 14]. Taking the *If* block as an example, although the false branch (i.e., the zombie region) is not executed, a default output value is still produced, which is not the case in traditional programming languages. On the other hand, the CPS language has an explicit notion of sampling time inference and data type inference. Such notion could cause zombie regions to vary with sampling times. For example, the inputs of CPS models are typically obtained from sensors, which have a fixed sampling time frequency (such as 10 times per second). Therefore, the output of the same block may change at different sampling times. This affects the subsequent control and data flows, which also changes the subsequent zombie regions.

For example, as shown in Figure 1, the *Sine Wave* block outputs values between -1 and 1 during the execution of the CPS model. Assuming that in the current sampling time, the *Sine Wave* block outputs 1. The *If* block then selects Action1 as the true branch and Action2 as the false branch (i.e., the zombie region). However, at the next sampling time, the *Sine Wave* block outputs -1; hence, Action1 becomes the false branch and Action2 is the true branch. Moreover, although a zombie region (i.e., the false branch) is not executed, it still outputs a default value (e.g., 0), which may affect the control and data flows of the CPS model.

The state-of-the-art testing methods for Simulink compiler testing are SLforge [13] and SLEMI [14]. SLforge is a CPS model generator, which is similar to Csmith [48]. Specifically, SLforge can generate a large number of random valid CPS models to test Simulink compiler based on some predefined configuration parameters [13]. Given a CPS model generated by SLforge, the study of SLforge [13] removes all blocks in its zombie regions under all possible inputs to produce an equivalent CPS variant. In such a way, Simulink compiler bugs can be found by comparing the outputs of the CPS model and its variant. However, this mutation strategy does not insert or delete any block from compilation regions (i.e., the live regions that can be compiled), meaning that it can only produce one equivalent variant for each seed CPS model. In contrast, SLEMI first takes a seed CPS model

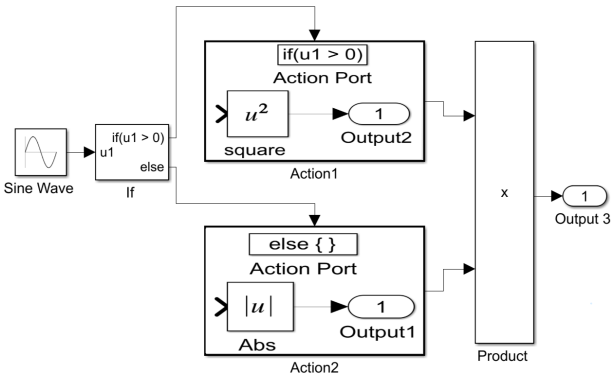

Figure 1: Example of a valid Simulink CPS model

generated by SLforge as input, and finds the zombie regions in this CPS model according to the data obtained during variable profiling. SLEMI proposes three mutation strategies to generate equivalent CPS variants for differential testing, namely, randomly deleting blocks in the zombie regions, replacing zombie regions with the *Saturation* block, and extracting blocks in zombie regions and promoting them to their own child model [14]. Therefore, SLEMI can only generate a limited number of CPS variants with low variation diversity. Specifically, two challenges remain to be addressed for Simulink compiler testing.

Challenge 1. The limited variant space. SLEMI only considers a limited number of zombie regions in the seed CPS model as the variant space to conduct mutation. When a seed CPS model contains few zombie regions, the mutation cannot be effectively applied. Since EMI-based differential testing relies on equivalent CPS models with different control and data flows to test various optimization strategies of Simulink compiler [25], this limitation can hinder their capability of testing Simulink compiler thoroughly.

Challenge 2. The insufficient mutation diversity. Although SLforge and SLEMI have demonstrated their bug-finding capability in Simulink compiler, their simple mutation strategies can result in insufficient diversity of CPS variants. Therefore, the bugs they detected can be saturated. Their simple mutation strategies may only trigger shallow bugs that are very close to the seed CPS models. Moreover, these similar CPS variants may trigger duplicate bugs.

To overcome these challenges, we propose a new method for Simulink compiler testing named **COMBAT** (**C**ontrollable z**OM**bie **B**locks mut**AT**ion) to generate equivalent and diverse CPS variants to exercise Simulink compiler thoroughly. COMBAT has an EMI mutation component and a diverse variant generation component. Specifically, the EMI mutation component first profiles variable valuations (i.e., the value range of each variable) at all CPS points of the seed CPS model. Then, COMBAT inserts assertion statements (e.g., *If*/*While* blocks) at arbitrary points between two blocks in the seed CPS model. According to the variable ranges obtained from variable profiling, COMBAT ensures these assertion statements always evaluated as true. Since an assertion statement breaks a CPS point into true and false branches, we feed all the data passed through this CPS point into the true branch to preserve the equivalence of CPS variants, and consider the body of

the false branch as the variant space. In this way, new variant space is generated to overcome **Challenge 1**. The diverse variant generation component uses the Markov chain Monte Carlo (MCMC) optimization sampling strategy to sample the seed CPS models, which generates complex mutations of long sequences of blocks in the variant space of the seed CPS model to detect deep bugs, allowing us to overcome **Challenge 2**. Moreover, the blocks are sampled in the seed CPS models, which could improve the accuracy of sample time inference and data type inference by Simulink compiler, thereby could improve the mutation effectiveness of COMBAT. Finally, COMBAT utilizes differential testing to validate all CPS variants. If the outputs of a CPS variant are different from the seed CPS model, the corresponding CPS variant is deemed to trigger a bug.

To evaluate the effectiveness of COMBAT, real-world CPS models [15] and CPS models generated by SLforge are used as seed CPS models to generate CPS variants. Our evaluation demonstrates that COMBAT significantly outperforms the state-of-the-art methods (i.e., SLforge and SLEMI) in terms of the bug-finding capability for Simulink compiler testing. Specifically, COMBAT can detect 6 bugs in four weeks, of which 2 bugs are new and 4 bugs are known, while SLEMI detects 3 bugs, of which 1 bug is new and 2 bugs are known. SLforge can only detect 1 known bug. Within five months, we have reported 16 valid bugs for the recently released version Simulink R2021b, of which 11 bugs have been confirmed as new bugs. In addition, the mutation effectiveness of COMBAT outperforms that of SLEMI by 183.33% and 220.83% in terms of the average number of newly added blocks and connections in CPS variants, respectively.

In summary, the main contributions of this work are as follows:

- We propose COMBAT to detect bugs in Simulink compiler. COMBAT utilizes an EMI mutation component and a diverse variant generation component to address the limited variant space and the insufficient mutation diversity challenges in Simulink compiler testing, respectively.
- Extensive experiments are conducted to evaluate the bug-finding capability of COMBAT. COMBAT has detected 16 valid bugs, of which 11 bugs have been confirmed as new bugs by MathWorks Support.
- We release COMBAT as a replication package for Simulink compiler testing [16].

Paper Organization: Our motivations are discussed in Section 2. The main components of COMBAT are introduced in Section 3. Experimental setups and results are presented in Sections 4 and 5, respectively. We elaborate on threats to validity in Section 6. Related works are discussed in Section 7. Finally, Section 8 presents the conclusion and future works.

2 BACKGROUND AND MOTIVATION

In this section, we first introduce the process of developing a CPS model in Simulink. Then, we use two bug examples to motivate and illustrate our new method COMBAT.

2.1 Preliminaries

Simulink is a block diagram environment for model-based design, which is the de-facto standard in safety-critical domains [22, 32].

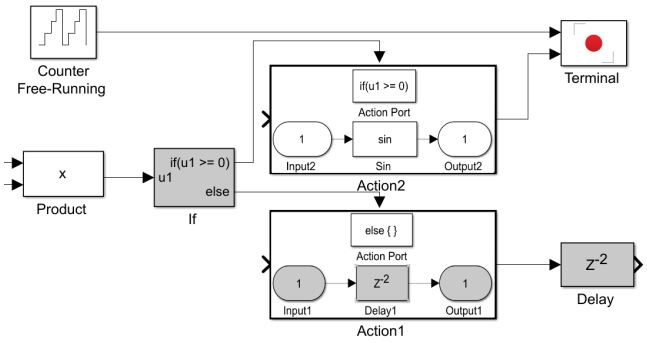

Figure 2: A reduced variant that triggers a sample time inference exception in Simulink R2021b in the *Normal* simulation mode

Simulink supports dataflow and object-oriented programming to rapidly prototype engineers' systems by modeling simulation and compiling executable. In the modeling simulation stage, engineers develop CPS models by using blocks and connection lines. The blocks perform some operations on data received by input ports, and then output calculation results through output ports. The connection lines pass these outputs to their subsequent blocks [13, 14, 23, 36]. On the connection lines, there are CPS points which can calculate variable valuations between two blocks. When a CPS model is designed, in the compiling executable stage, Simulink compiler conducts the syntax/semantics analysis, and automatically infers datatypes of blocks for the CPS model, as well as CPS model optimization to accelerate software upgrades. After the successful compilation, an executable is generated, which is commonly deployed on the target hardware for safety-critical applications, such as aerospace and healthcare [20, 22, 35, 50].

In the aforementioned process, Simulink compiler bugs may cause the functionality of the developed CPS model beyond engineers' expectations. Generally, there are two main types of Simulink compiler bugs during compilation, namely crashes and miscompilations. Crash bugs cause Simulink to crash during design, modeling, and compilation, which manifest as parse errors or internal assertion failures. In contrast, miscompilation bugs lead to inconsistent behaviors between the developed CPS model and engineers' expectations. Furthermore, miscompilation bugs are difficult to recognize because they typically manifest indirectly as CPS model failures. Since CPS models developed by Simulink are often deployed in safety-critical environments, it is crucial to ensure the correctness of Simulink compiler.

2.2 Illustrative Examples

Two confirmed bug examples are discussed in this subsection to motivate and illustrate COMBAT. The first bug is triggered by the EMI mutation component, which demonstrates how we profile variable valuations of the seed CPS model to add new assertion statements (e.g., *If*/*While* blocks). The added assertion statements preserve the equivalence of CPS variants with different control and data flows to test various optimization strategies in Simulink compiler. The second bug is triggered by the diverse variant

generation component, which samples blocks derived from the seed CPS model to perform a sequence of mutations in the variant space.

In this study, we only present the reduced versions of bugs[1] because the original seed CPS models are too large for presentation (with over hundreds of blocks).

Technical Support Case (TSC) 05274593: *Simulink compiler miscompilation error*. Figure 2 presents the reduced variant that triggers a Simulink compiler miscompilation bug. The sequence of blocks inserted by COMBAT is highlighted in grey. This variant is generated as follows:

(1) COMBAT profiles the variable valuations at all CPS points of the seed CPS model (i.e., the CPS model in Figure 2 excluding the blocks in grey). For example, COMBAT could find that the value range of variable u_1 is $[0, +\infty)$.

(2) COMBAT adds an assertion statement on the CPS model. In this example, we add an *If* block. According to the value valuation of u_1, we can enforce the conditional predicate of this *If* block always evaluated as true. Hence, the output u_1 of the *Product* block can be directly passed to the *Sin* block in the true branch to preserve the equivalence of the CPS variants. Since the false condition of the *If* block is never executed, the inserted *If* block does not have any side effects on the CPS model, which can be considered as the EMI variant space. COMBAT can insert new blocks (such as the Action1 block in grey) in the variant space.

When compiling this CPS variant, Simulink R2021b throws a sample time inference exception in the *Normal* simulation mode. However, the CPS variant is equivalent to the seed CPS model. Since the output of the *Product* block is greater than zero under all inputs, the false branch of the *If* block is never executed. Simulink compiler should infer the sample time of the blocks normally, no matter what blocks are inserted in the false branch.

However, this CPS variant triggers a bug because Simulink compiler incorrectly assumes that the access to the zombie region (i.e., *Action1* and *Delay*) are non-trapping; it elevates the *Delay* blocks out of the false branch of the *If* block, which causes the CPS variant to be executed unconditionally. Specifically, Simulink compiler incorrectly infers the sample time of the terminal block, and makes it as a multi-rate block. Existing methods (e.g., SLforge and SLEMI) cannot change the control and data flows of the seed CPS model to reveal this bug.

TSC 05314520: *Simulink compiler compilation crash*. Figure 3 presents a reduced variant that triggers a Simulink compilation crash bug when compiling the model with Simulink R2021b in the *Normal* simulation mode. Initially, Simulink successfully compiles the seed CPS model. We use the diverse variant generation component of COMBAT to sample the CPS model, and generates complex mutations of long sequences of blocks in the variant space (i.e., the *Action2*) of the seed CPS model. Note that the variant in Figure 3 has already been reduced. The original CPS model and its variants can be found in our replication package [16]. Since the generated blocks are in the variant space (i.e., the zombie region), the blocks in this space (i.e., the *Action2*) should not be executed. However, after COMBAT generates a sequence of blocks, Simulink

[1]A reduced version of a bug-triggering CPS model can be derived by removing blocks that are not related to the bug. It helps Simulink developers understand and fix bugs.

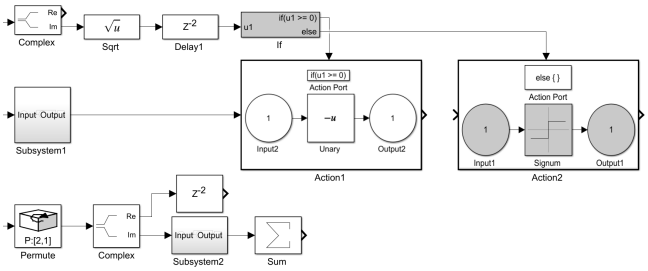

Figure 3: A reduced CPS variant that triggers a Simulink compiler compilation crash bug in Simulink R2021b in the *Normal* simulation mode

R2021b crashes. Specifically, the data type of *Complex* block is incorrectly inferred with and without logging the signal in this CPS variant. Before recording the signal, the signal type of the *Complex* block is double. However, after recording the signal, the Simulink compiler incorrectly infers the type of the *Complex* block as unit32. Eventually, the heuristic inference of the entire CPS model fails and triggers a Simulink compilation crash bug.

This Simulink compilation crash bug was not triggered in a simple mutation, but by a sequence of mutations. SLforge and SLEMI cannot reveal this bug because they cannot generate complex mutations with long sequences of blocks in the variant space.

3 COMBAT FRAMEWORK

In this section, we first present an overview of the framework of COMBAT. We then explain the EMI mutation component and the diverse variant generation component to address the two challenges. Finally, the details of differential testing are presented.

3.1 Overview

The framework of COMBAT is illustrated in Figure 4 and Algorithm 1. COMBAT consists of three components: the EMI mutation component, the diverse variant generation component, and the differential testing component. The basic idea of COMBAT is to generate diverse and equivalent CPS variants to exercise Simulink compiler thoroughly with differential testing. Specifically, the EMI mutation component (lines 10–15) profiles variable valuations of the CPS model and inserts assertion statements (e.g., *If*/*While* blocks) that always evaluate to be true at arbitrary points between two blocks in the seed CPS model. Hence, the outputs of the blocks before the assertion statements can be directly passed to their subsequent blocks in the true branches to preserve the equivalence of the CPS variants. We consider the body of the false branches that are never executed as the variant space (Figure 4(a)). Hence, new variant space with different control and data flows is generated to address the limited variant space challenge. The diverse variant generation component (lines 16–24) uses MCMC optimization sampling strategy to effectively sample the CPS model (Figure 4(b)). It generates complex mutations of long sequences in the variant space to detect deep bugs to overcome the insufficient mutation diversity challenge. Finally, COMBAT utilizes differential testing (lines 3–9) to validate all CPS variants (Figure

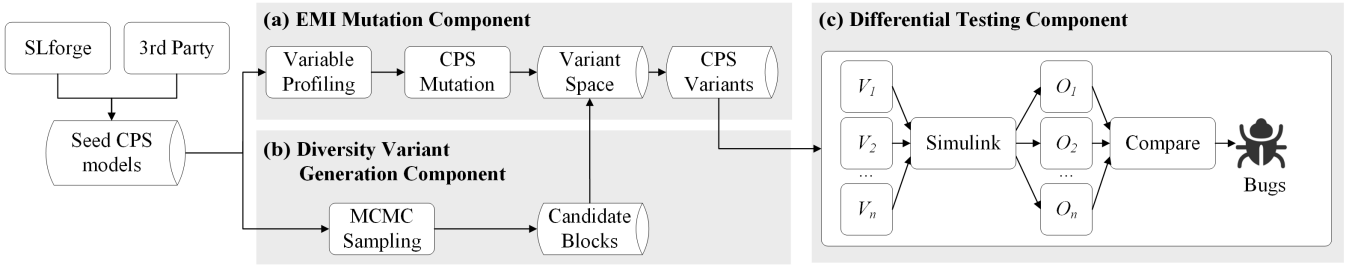

Figure 4: The framework of COMBAT

Algorithm 1 COMBAT

Input: Simulink S, Seed CPS model C, CPS parameters P, Mutant number MAX-ITER
Output: Reported Bug

```
1:  procedure TEST(S, C, P)
2:      O ← S.Compile(C).Execute(P)
3:      for MAX-ITER do
4:          C′ ←EMI(C, P)
5:          O′ ←S.Compile(C′).Execute(P)
6:          if O′ ≠ O then
7:              Return ReportBug(S, C′, P)
8:          end if
9:      end for
10:     function EMI(CPS C, CPS parameters P)
11:         V ←Profile(C, P)
12:         a ←GenAssertion(C, P, V)
13:         C′ ←Mutate(C, a)
14:         Return C′
15:     end function
16:     function MUTATE(CPS C, Assertion statement a)
17:         B ← Statistics(C, a)
18:         vs ←GetVariantSpace(C, a)
19:         if B ≠ ∅ then
20:             bs ←MCMC(C, B)
21:             C′′ ← C.Insert(vs, bs)
22:         end if
23:         Return C′′
24:     end function
25: end procedure
```

4(c)). If the output of a CPS variant is different from the seed CPS model, it is deemed to trigger a Simulink compiler bug.

The framework of COMBAT is inspired by the recent works for compiler testing [26, 44]. We adopt and improve the existing testing framework based on the unique CPS language characteristics (e.g., the notion of sample time inference and datatype inference).

3.2 EMI Mutation Component

The EMI mutation component includes two sub-components, i.e., variable profiling and CPS mutation.

Variable Profiling. COMBAT begins by profiling the variable valuations (i.e., the value range of each variable) at all CPS points of the seed CPS model under all inputs. We implement the variable profiling with the Signal Range Coverage tool in Simulink [1]. Figure 2 can be considered as an example. COMBAT profiles the seed CPS model (i.e., the CPS model in Figure 2 excluding the highlighted blocks in grey). During variable profiling, for example, the value range of variable u_1 is $[0, +\infty)$.

CPS Mutation. In this step, COMBAT randomly selects one or more CPS points in the seed CPS model. COMBAT inserts an assertion statement to each selected CPS point. All assertion blocks in CPS models that generate new control/data flows can be inserted, such as *If*/*While* blocks, *Function-Both Call Split* blocks, and *For Iterator Subsystem* blocks. According to the variable profiling [2], COMBAT sets the predicate of each assertion statement always to be true. For example, COMBAT sets the predicate of the *If* block in Figure 2 as $u_1 \geq 0$. By this setting, the output of the block before the assertion statement can be directly passed to its subsequent blocks through the true branch to preserve the equivalence of the CPS variants. We consider the false branch that is never executed (i.e., the zombie region) as the variant space, thus creating new variant space for generating CPS variants.

Compared to SLforge and SLEMI, COMBAT can significantly increase the variant space by inserting assertion statements at any point in the seed CPS model. This strategy increases the control and data flows of the generated equivalent CPS variants. When Simulink compiler compiles a CPS model, it needs to determine which optimization methods are applicable to the current control and data flows. Therefore, COMBAT can exercise Simulink compiler more thoroughly by forcing it to use various optimization methods on CPS variants.

We remark that inserting assertion statements is a common strategy to maintain the EMI property [44, 47]. COMBAT adopts this strategy for Simulink compiler testing by considering two main differences. First, CPS models have sample time. Variable values can change significantly at different sample time points, e.g., ranging $[0, +\infty)$. Different from existing studies [44], COMBAT generates true/false predicates by analyzing the continuous or discrete ranges of variables instead of a single variable value. Second, due to the automated datatype inference by Simulink, a small mutation can trigger vastly different inferred datatypes and break the EMI property [14], which is not the case for C/C++ compilers. Hence, we record the inferred datatypes of blocks in the seed CPS model and add a *Datatype Conversion* block before the

corresponding block in CPS model variants to maintain the datatype equivalence.

3.3 Diverse Variant Generation Component

The diverse variant generation component first computes the basic statistics of blocks in the CPS models (e.g., the number of blocks in the seed CPS model). Then, MCMC optimization sampling strategy is used to sample the blocks in the CPS model effectively and insert new blocks into the variant space. This component aims to generate complex mutations with long sequences of blocks in the variant space to detect deep bugs. The basic idea of MCMC optimization sampling strategy is to construct a Markov chain based on the probability distribution and the transition probability of blocks in the seed CPS model. This strategy uses the Markov chain to decide which blocks should be generated in the variant space.

Specifically, when a block b is sampled in the variant space, the probability to accept the block $\hat{b}$ as the next block to be generated in the variant space is defined as follows:

$$A(b \rightarrow \hat{b}) = \min(1, \frac{\theta(\hat{b})q(b|\hat{b})}{\theta(b)q(\hat{b}|b)}), \quad (1)$$

where $\theta(\hat{b})$ is the probability density distribution of block $\hat{b}$ in the seed CPS model and $q(\hat{b}|b)$ represents the one-step transition probability from b to $\hat{b}$.

We calculate the probability density distribution of each block as the ratio of this block in the seed CPS model. We generate blocks in the variant space by sampling existing blocks in the seed CPS model. The sampling strategy assumes that blocks which have been frequently used should have a higher probability to be selected, since it ensures that the sampled blocks have a similar distribution with existing valid CPS models. To calculate the transition probability, a transition matrix is typically constructed to record the one-step transition probability between different blocks. To obtain the transition matrix, we use the variable control method to compute the transition probability between a selected block and its child blocks. Specifically, given a block in the seed CPS model, we consider this block as the root block in the tree structure and the blocks connected to it as the child blocks. We then use the normalized frequency of the connection between the root block and a child block to represent their one-step transition probability. That is, if a block is closer and frequently connected to the selected root block, it is more likely to be sampled after the root block. It ensures that when generating a sequence of blocks in the variant space, the sequence of blocks does not exhibit unexpected behavior due to errors in sampling time inference and data type inference.

After the probability density distribution and the transition probability are calculated, we conduct our sampling strategy. Initially, we randomly generate a block in the variant space. Based on this block, we calculate the acceptance probability of each block as the next block with formulae 1 and generates the next block based on this probability. We continue the sampling step until the number of blocks generated in the variant space reaches an output block. In this way, we can generate complex mutations of long sequences of blocks in the variant space to detect deep bugs, which can overcome the insufficient mutation diversity challenge.

Algorithm 2 MCMC Optimization Sampling Strategy

Input: CPS models C, Blocks B
Output: diverse variants V
1: $\theta(b) \leftarrow$ Parse(B) //probability density
2: $q(\hat{b}|b) \leftarrow$ Count(C) // transition probability
3: $\hat{b} \leftarrow$Init()
4: **while** $\hat{b} \neq$ output block **do**
5: $\hat{b} \leftarrow$Sample(B, $q(\hat{b}|b)$)
6: Calculate acceptance probability $A(b \rightarrow \hat{b})$)
7: $Z \leftarrow$Randomly $Uniform[0, 1]$
8: **if** $z \leq$ A(b$\rightarrow\hat{b}$) **then**
9: $V \leftarrow V \cup \hat{b}$
10: **end if**
11: **end while**
12: Return V

Algorithm 2 describes the main process of the MCMC optimization sampling strategy. We first parse the seed CPS model to obtain all block types and the probability density $\theta(b)$ on line 1 and count the transition probability $q(\hat{b}|b)$ for each block b to the next block $\hat{b}$ on line 2. We then use MCMC optimization to sample the CPS model space to generate additional variants (lines 4 to 6). We perform sampling judgment during MCMC sampling. We randomly sample z from the uniform distribution (0, 1) and compare it to the acceptance probability to determine whether to accept the block $\hat{b}$ (lines 7 to 10) . The sampling stops, when the sampled block reaches the output block.

The MCMC sampling strategy used by COMBAT is inspired by a recent C/C++ compiler testing tool Athena [26]. However, the CPS language has its unique characteristics. First, as a block-based language, to improve the variant diversity, COMBAT uses the MCMC sampling strategy to generate block sequences rather than reusing existing code snippets directly [26]. Second, Athena [26] selects code snippets by a program distance objective function. In contrast, COMBAT improves the diversity by generating different and valid block combinations. Third, COMBAT solves the datatype compatibility problem by the block transition probability and Simulink datatype inference, while Athena [26] uses a code snippet context table to maintain the datatype consistency.

3.4 Differential Testing Component

Differential testing for Simulink compiler is different from existing EMI-based compiler testing approaches for procedural programs [25, 26, 44]. In compiler testing, the test program has only one output. In contrast, CPS models have a clear sampling time characteristic, that is, the output of the same block can be different at different sampling times. Therefore, in the differential testing phrase, we compare the data values of all blocks for every sampling time within a time interval. In our differential testing, we run every CPS variant in both *Normal* and *Accelerator* simulation modes. If the output of a CPS variant is different from the output of the seed CPS model in any simulation mode, the corresponding CPS variant is considered to trigger a Simulink compiler bug.

4 EVALUATION

In this section, four experiments are conducted to evaluate the effectiveness of COMBAT. Specifically, our evaluation aims at answering the following Research Questions (RQs).

RQ1: How is the bug-finding capability of COMBAT compared to the state-of-the-art methods?

RQ2: Can COMBAT detect new Simulink compiler bugs?

RQ3: How effectiveness is the mutation strategy of COMBAT compared to SLEMI?

RQ4: How effectiveness is the MCMC optimization sampling strategy of COMBAT?

In our experiments, RQ1 and RQ2 are used to evaluate the bug-finding capability of COMBAT compared to the state-of-the-art approaches. RQ3 and RQ4 are employed to evaluate the mutation strategy of the EMI mutation component and the diverse variant generation component.

4.1 Seed CPS Models

To evaluate the effectiveness of COMBAT, both real-world CPS models and CPS models generated by SLforge are used as seed CPS models, according to the existing study of SLEMI. Specifically, the real-world CPS models are collected from a large corpus of 1,000 publicly available CPS projects created by engineers [15]. In this corpus, we filter the CPS models that cannot be run in our Simulink runtime environment. At last, 357 CPS models are remained. We use these real-world models because it means that the bugs we found are likely to directly affect engineers. Since the number of real-world CPS models is small, following the previous study [14], we also add CPS models generated by SLforge to the seed CPS model set. SLforge is a widely used CPS model generator that supports most features of the CPS language specification. As suggested in the existing study [14], the size of the seed CPS models generated by SLforge is between 100 and 3,000 blocks. With these seed CPS models, we use COMBAT to generate CPS variants for Similink compiler testing.

4.2 Evaluation Setup

COMBAT is implemented in MATLAB. The source code and experimental data are available at GitHub [16]. Our evaluation is run on a computer with Windows 10 64-bit system, an Intel Core i9 CPU@2.10 GHz, and 120 GB of memory.

We evaluate COMBAT and the baselines as follows. Given an evaluation period (e.g., two weeks), we first feed the real-world seed CPS models to each algorithm to generate CPS variants for detecting Simulink compiler bugs. After all these models have been used, we continue using SLforge to generate additional seed CPS models for each algorithm until the end of the evaluation period. Regarding the parameters, we set the number of CPS variants for each seed CPS model (i.e., MAX-ITER in Algorithm 1) to 5, which is the same as SLEMI. We use the default parameters of SLforge [13] and SLEMI [14].

When a Simulink compiler bug is detected, similar to the related works [24, 44, 45], we reduce the CPS model that triggers the bug before reporting them to the MathWorks's bug reports website, such that Simulink developers can quickly understand and fix bugs. Specifically, we try to remove blocks in the CPS model one by one. If the bug still exists after removing a block, we consider this block as no effect on the bug. Otherwise, the removed block is put back in its original place. This process continues until no block can be removed. To avoid reporting duplicate bugs, we use the failed assertion and back-trace to detect duplicate bugs before bug submission. We treat two bugs as duplicate bugs when they have the same failed assertion or back-trace.

We submit an issue for each bug to the MathWorks's bug reports website [3]. Then, the MathWorks Support will communicate the specifics on the issues by email, and finally classify each issue into new/known/nonbug/pending. Unlike open source projects that typically list all bug reports, the MathWorks's bug reports website only lists bug reports found in the early stage. Meanwhile, this list is neither comprehensive nor mentioning the corresponding TSC number of each issue. To make it easier to reproduce our bugs, we release the CPS models that trigger bugs on GitHub [16].

We compare COMBAT against SLforge and SLEMI. They are two state-of-the-art methods for Simulink compiler testing. We reproduce SLforge and SLEMI with source code provided by their works and use their default configurations.

4.3 Answer to RQ1

Approach. To evaluate the effectiveness of COMBAT, we compare the bug-finding capability of COMBAT with the state-of-the-art methods SLforge [13] and SLEMI [14], since finding more bugs within a time period is the main objective of these approaches. In the experiment, we detect bugs on both the recently released version Simulink R2021b and a previous version Simulink R2018a, since Simulink R2018a has been used for evaluation in SLforge and SLEMI. We set a single testing period of two weeks for each Simulink version, that is, every method tests Simulink R2018a and R2021b for two weeks.

Results. As shown in Table 1, bugs detected in the experiment can be classified into new bugs (*New*) and known bugs (*Known*). Bugs labeled as *Known* means they are duplicate with the bugs in the bug repository. It is obvious from Table 1 that COMBAT significantly outperforms SLforge and SLEMI in terms of the bug-finding capability. COMBAT can detect 6 bugs in four weeks on the two Simulink versions, of which 2 bugs are new and 4 bugs are known, while SLEMI detects 3 bugs in total, of which 1 bug is new and 2 bugs are known. SLforge can only detect one known bug. Moreover, the relationship of bugs detected by SLforge, SLEMI and COMBAT in Simulink R2018a and Simulink R2021b can be seen in Figure 5. The reason is that COMBAT can generate new variant space with different control flow and data flow characteristics. When Simulink compiles a CPS model, the compiler uses static analysis to determine which optimization methods are applicable to the current control flow and data flow of the CPS model. COMBAT can exercise Simulink compiler more thoroughly by forcing it to use various optimization methods on variants. Moreover, COMBAT uses MCMC optimization to sample the seed CPS model effectively, which generates complex mutations of long sequences in the variant space. We can see that the number of bugs detected in Simulink R2018a is more than in Simulink R2021b. This is because of the fact that the recently

Table 1: Bugs detected by SLforge, SLEMI and COMBAT in Simulink R2018a and Simulink R2021b

	Simulink R2018a		Simulink R2021b		Total
	New	Known	New	Known	
SLforge	1	0	0	0	1
SLEMI	1	1	0	1	3
COMBAT	1	3	1	1	6

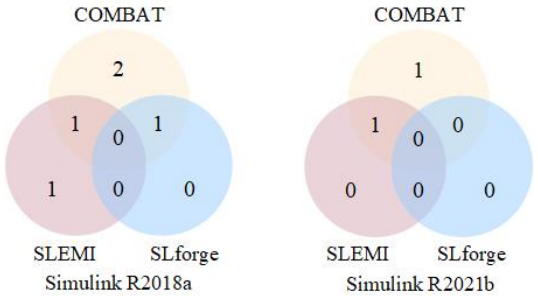

Figure 5: The relationship of Bugs detected by SLforge, SLEMI and COMBAT in Simulink R2018a and Simulink R2021b

released version 'Simulink R2021b' is constantly maintained by the Simulink developers.

In addition, we find that the bugs detected by SLforge and SLEMI are a subset of the bugs detected by COMBAT, since SLforge and SLEMI are limited by the variant space and their simple mutation strategy. Specifically, COMBAT can significantly increase the variant space based on its ability to insert assertion statements at any point in the seed CPS model. For example, by generating blocks in the newly added variant space, COMBAT detects the bug **TSC 05274593**. This bug occurs because Simulink compiler incorrectly assumes that the access to the zombie variant space is non-trapping, so it elevates the blocks in this region out of the false branch of the *If* block. This operation causes the CPS model to be executed unconditionally. As a result, Simulink compiler incorrectly infers the sample time of the terminal block as a multi-rate block. SLforge and SLEMI failed to reveal this bug because they cannot change the control flow and data flow of the seed CPS model to add new variant space.

Conclusion. COMBAT significantly outperforms SLforge and SLEMI for testing Simulink compiler. These results verify the bug-finding capability of COMBAT.

4.4 Answer to RQ2

Approach. We conduct an experiment over five months from November 2021 to March 2022 to evaluate the bug-finding capability of COMBAT in practice. In this experiment, we test the recently released version of Simulink (i.e., R2021b), since Simulink developers fix bugs primarily in the recently released version rather than in old versions. We submitted all the detected bugs as issues to the MathWorks's bug reports website [4].

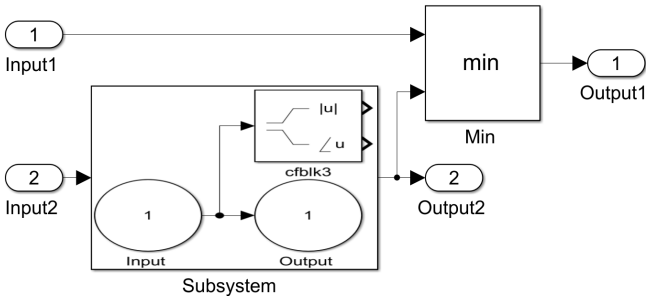

Figure 6: TSC 05310042. Math function error in accelerate moode by selecting *NaN* and One

Results. In five months, we have reported in total 16 unique issues, of which 11 have been confirmed as new bugs and 5 are already known to MathWorks. Among them, 15 of the seed CPS models that triggered the bugs are generated by SLforge, and 1 is from the real-world models (i.e., **TSC 05382872**). This is due to the limited number of real-world models (i.e., 357 seed CPS models). In contract, the number of seed CPS models generated by SLforge is large, and COMBAT can generate more equivalent and diverse variants to exercise Simulink compiler thoroughly. Table 2 summarizes the reported bugs. Each bug has a unique TSC number from MathWorks. There are two types of status of the feedback from MathWorks on bug report (i.e., *New* = newly confirmed bug, *Known* = known bug). There are two types of bugs in our reported bugs, including crash bugs (*C*) and miscompilation bugs (*M*).

From Table 2, we find that 8 crash bugs and 8 miscompilation bugs have been confirmed or fixed by MathWorks Support. Crash bugs cause Simulink to report parse errors or internal assertion failures. For example, **TSC 05314520** is a bug found by COMBAT. This is the bug illustrated in Figure 3. In this bug, COMBAT generates a series of blocks in the variant space, which causes the data type of the *Complex to Magnitude-Angle* block to be incorrectly inferred. As a result, Simulink R2021b compiler crashes in the *Normal* simulation mode. Compared with crash bugs, miscompilation bugs in Simulink are harder to detect because they often manifest indirectly as CPS model failures compared to more frequent bugs in CPS models. **TSC 05310042** is an incorrect code generation bug, and it causes the *Min/Max* blocks produce incorrect results in the *Accelerator* simulation mode as shown in Figure 6. Specifically, the *Min* block has only one input and the first element of the input is *NaN*. This is problematic because input [*NaN*, 1] can output *NaN*, which is an incorrect result. This bug is fixed[2] only after two weeks since it is a critical bug which affects Simulink from R2015 to R2021.

Existing approaches (i.e., SLforge and SLEMI) may not reveal these bugs for two reasons. First, COMBAT generates new variant space with different control/data flows to test different optimization methods of Simulink compiler. Existing approaches seldom change the control/data flow. Second, COMBAT uses the MCMC sampling strategy to generate complex block sequences, while SLforge and SLEMI mainly delete/modify existing blocks. This is also confirmed

Table 2: Bugs reported by COMBAT

TSC	Summary	Status	Type
05255309	Data type error cause cannot open or compile the file normally	New	C
05274593	Abnormal If block condition judgment	New	C
05294630	Math function error in accelerate by selceting NaN and zero	New	M
05296099	Data error for signal generator module in accelerated mode	New	M
05310042	Min module data inconsistency in accelerate simulation	Known	M
05314517	Reference model inheritance time exception	New	C
05314520	The data type error after logging the signal is inconsisten	New	C
05317645	Data type judgment error under reference model in Math operation module	New	C
05320137	Exception output of abs block in accelerated compile mode	Known	M
05358090	Reference model sampling time inference exception	New	M
05358093	Heuristic inference exception in logging signal	New	C
05371387	Zero-cross detection cause data exception	New	M
05382872	Compile error after logging signal in complex module	New	C
05382877	Accelerate simulation compile errors by using Lcc	Known	C
05398645	Max module misbehaves under zero-crossing detection	Known	M
05405356	Abnormal MAX zero-crossing detection in acceleration mode	Known	M

[1] There are two types of status feedback from MathWorks on bug report (i.e., New = newly confirmed bug, $Known$ = known bug). There are two types of bugs (i.e., Type) in our reported bugs: crash bugs (C) and miscompilation bugs (M).

by RQ1, where SLforge only finds one bug and COMBAT finds several bugs missed by SLEMI.

Conclusion. COMBAT is effective in detecting Simulink bugs. Within five months, we reported 16 valid bugs, of which 11 bugs have been confirmed as new bugs.

4.5 Answer to RQ3

Approach. In this experiment, we compare the mutation effectiveness of COMBAT with SLEMI by analyzing the properties of variant CPS models (such as the number of newly added blocks, connections, and assertion statements). We do not evaluate SLforge because it only uses a simple strategy (i.e., deleting all dead blocks in the zombie region) to create variant CPS models. We count the total CPS variants generated by COMBAT and SLEMI during one week for testing Simulink R2021b. The box-plots in Figure 7 show the collected three metric values, including the number of newly added blocks and connections in the variant space, and assertion statements in each variant CPS model.

Results. Number of newly added blocks and connections. The most important elements of CPS models are blocks and connections, which are widely counted for representing the characteristics of CPS models[30, 34]. According to [13, 37], the implicit connections (e.g., the input or output of the connection is incomplete) are not included in the connection-count metric. As shown in Figure 7, the minimal and maximal numbers of the newly added blocks and connections in CPS model generated by COMBAT are [21, 223] and [20, 210] respectively, which are larger than those of CPS variants generated by SLEMI. Specifically, Figure 7 illustrates that COMBAT outperforms SLEMI by 183.33% and 220.83% in terms of the median of the first four metric values. The results show that the scale and diversity of the CPS variants generated by COMBAT are superior to that of SLEMI.

[2]https://ww2.mathworks.cn/support/bugreports/2648231?ref=ts_R2022a_Update_1.

Number of assertion statements. The effectiveness of EMI depends on creating new variant space by altering the control and data flows of the seed CPS model. This characteristic helps test various optimization strategies of Simulink [25]. As shown in Figure 7, the minimal and maximal numbers of assertion statements in variants generated by COMBAT are larger than those generated by SLEMI. COMBAT outperforms SLEMI by 16.67% in terms of the median of the number of assertion statements.

Conclusion. The mutation effectiveness of COMBAT outperforms SLEMI. COMBAT can generate diverse CPS variants with different control and data flows, which can partially explain the reason why the bug-finding capability of COMBAT is better than SLEMI in most cases.

4.6 Answer to RQ4

Approach. To evaluate the effectiveness of sampling generative variants by MCMC optimization sampling strategy, we design an experiment (i.e., keep the variant space of CPS variants consistent) for comparing the mutation effectiveness in variant space of the MCMC optimization sampling strategy with the random selection strategy and the database search strategy, which are commonly used selection strategies. The random selection strategy randomly selects blocks from the mathtool [23], while the database search strategy randomly selects blocks from the existing seed CPS model. The experiment is conducted on Simulink R2021b within one week. The number of newly added blocks and connections by different strategies in CPS variants and the successful mutation rate are used to evaluate the mutation effectiveness.

Results. As shown in Table 3, the successful mutation rate in variants by MCMC optimization sampling strategy is 97.67%, which is clearly larger than that of the random selection strategy and the database search strategy (i.e., 31.15% and 69.12%). Moreover, from Table 3, we find that the average numbers of newly added

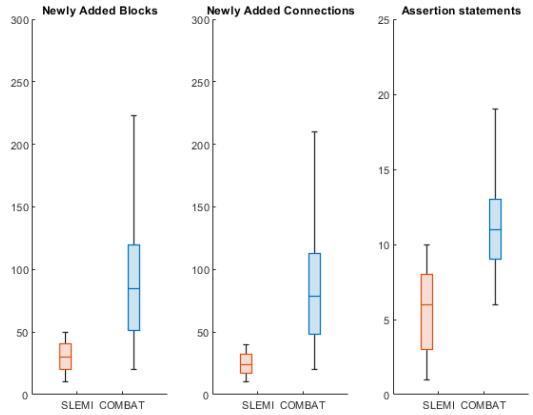

Figure 7: The distribution of generated CPS variants by SLEMI and COMBAT

Table 3: The mutation efficiency of different sampling strategies

	Blocks	Connections	Successful mutation
Random selection	187	167	31.15%
Database search	201	224	69.12%
MCMC sampling	211	191	97.67%

blocks and connections generated by MCMC optimization sampling strategy are 211 and 191, respectively. Although in some cases the number of connections generated by MCMC optimization sampling strategy is lower than the Database search strategy, the average numbers of blocks and successful mutation rate illustrates that the mutation effectiveness of MCMC optimization sampling strategy is better than the baselines in most cases. It means our strategy ensures that when generating a sequence of blocks in the variant space, the sequence of blocks does not exhibit unexpected behavior due to errors in sampling time inference and data type inference.

Conclusion. The MCMC optimization sampling strategy can effectively generate diverse CPS variants.

5 THREATS TO VALIDITY

Internal Threats. The main threat to the internal validity of our method is the correctness of COMBAT and the reproduced models SLforge and SLEMI. We double checked our code to reduce errors in COMBAT. Additionally, we reproduced SLforge and SLEMI by referencing the open-source code provided by their original papers. We have fully verified their accuracy. Furthermore, COMBAT only supports a subset of CPS language specifications and types of blocks because of the lack of complete formal language specifications. However, COMBAT has identified several confirmed Simulink compiler bugs using this subset of CPS language specifications. We will strengthen COMBAT to consider more types of blocks and user-created models in follow-up work to make our testing of Simulink compiler more complete.

External Threats. A key threat to the external validity of COMBAT is that it can trigger duplicate bugs during the testing process. To alleviate this threat, we reduce bug-triggering variants by removing blocks that are unrelated to bugs to help us understand and analyze bugs more accurately. According to the verification of MathWorks Support, the bugs we submitted are not duplicated with each other at present, which indicates that our method can alleviate this threat.

We generate CPS variants with COMBAT only in the latest version of Simulink R2021b to evaluate its bug-finding capability in practice. Compared to Simulink R2018a used by SLforge and SLEMI, Simulink R2021b is more functional and complex. To alleviate this threat, we compared the bug-finding ability of COMBAT with SLforge and SLEMI also on the same old Simulink version (i.e., Simulink R2018a) in RQ1.

6 RELATED WORK

6.1 Simulink Testing

There are two methods for testing Simulink compiler, namely SLforge and SLEMI [13, 14]. SLforge is a CPS model generator based on some predefined configuration parameters. In contrast, SLEMI was proposed to find the zombie regions of CPS models according to model profiling data. SLEMI proposes three mutation strategies to generate equivalent CPS variants for differential testing of Simulink compiler, namely, randomly deleting blocks in the zombie regions, replacing zombie regions with the *Saturation* block, and extracting blocks in zombie regions and promoting them to their own child model [14]. However, these methods are still have the limited variant space and the insufficient mutation diversity challenges. We propose COMBAT to address these limitations and better test Simulink compiler.

To tackle the lack of complete formal language specifications, Shrestha et al. [40] propose a framework DeepFuzzSL to learn validity rules automatically by the language model from the existing corpus of CPS models. In their subsequent work, in order to address the lack of CPS models, they use transfer learning to learn knowledge from both randomly generated models and models mined from open-source repositories to generate new CPS models for testing [41]. However, their experiments shown that the bug-finding capability of their methods are not strong, thereby, these two methods are not used as baselines in our experiments.

There are also works for testing [38, 42, 43] and analyzing [19] a selected part of Simulink. These works focus on different modules in Simulink aparts from Simulink compiler. For example, Fehér et al. [19] model the data type inference logic of CPS blocks in Simulink [24]. Stürmer et al. [42, 43] use graph grammars to test optimization rules of code generators. In contrast, we mainly focus on testing the simulation compilation part of Simulink.

6.2 Differential Testing in Compiler testing

Differential testing considers equivalent variants of test cases to generate variants according to the inputs of a program. It detects bugs by comparing the results of different variants [9, 29, 31, 49]. Differential testing has been fully verified in widely used compilation tools such as GCC. It has identified over 1,000 bugs [25, 26, 44, 45].

In compiler testing, there are three EMI-based difference testing mutation approaches, including Orion [25], Athena [26], and Hermes [44]. Orion focuses on mutating dynamically dead program regions by randomly pruning unexecuted statements to generate variant programs [25]. Athena can insert code into or delete code from dead code regions under different inputs [26]. Unlike Orion and Athena, which can only mutate in dead code regions, Sun et al. [44] can mutate both live and dead code regions to generate equivalent variants.

In addition to EMI-based differential testing, Jiang et al. [24] presented CTOS which uses arbitrary optimization sequences for identifying compiler bugs in LLVM. Their approach significantly increases the ability to detect bugs. Tang et al. [45] presented a diversity-guided program mutation method for detecting compiler-warning bugs. Chen et al. [11] proposed history-guided configuration diversification generation to test compilers to solve the singleness issue of test programs generated by the Csmith.

To accelerate compiler testing, Chen et al. [8] predict the probability that whether a test program can detect a bug. Based on this bug-revealing probability, all test programs are ordered and executed to achieve accelerated testing. Recently, Chen et al. [12] presented a method that predicts the test coverage of a test program for compiler testing and then accelerates compiler testing based on test coverage information.

However, existing differential testing methods cannot be directly applied to testing Simulink compiler. One key difference is that the CPS language specification has an explicit notion of sampling time, which gives rise to zombie regions in CPS models that do not exist in traditional programming languages. Furthermore, both true and false if-then-else branches will simultaneously return values, regardless of the selection of the *If* block, which is also dissimilar to traditional programming languages.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we propose COMBAT, an automated Simulink compiler bug detection approach. COMBAT includes an EMI mutation component and a diverse variant generation component, which address the limited variant space challenge and the insufficient mutation diversity challenge of Simulink compiler testing, respectively. COMBAT can generate a large number of CPS variants with different control and data flows to exercise the Simulink compiler more thoroughly by forcing it to use various optimization methods on CPS variants. Within five months, we have reported in total 16 unique issues on the latest version of Simulink (i.e., Simulink R2021b), of which 11 have been confirmed as new bugs by MathWorks Support.

In future work, we plan to further improve COMBAT by supporting more types of blocks and user-created models. In addition, we plan to conduct an empirical study to deeply compare the effectiveness and the types of detected bugs of different testing approaches on more CPS development tool chains.

ACKNOWLEDGMENTS

This work is supported in part by the National Natural Science Foundation of China under Grants 62032004, 61902050, 62132020, and Fundamental Research Funds for the Central Universities (NO.3132019355, NO.DUT22RC(3)028, NO.NJ2020022)

REFERENCES

[1] MathWorks. https://ww2.mathworks.cn/help/slcoverage/ug/types-of-model-coverage.html/ (last access: 16/03/2022).
[2] MathWorks. https://ww2.mathworks.cn/en/products/simulink.html/ (last access: 16/03/2022).
[3] MathWorks. https://www.mathworks.com/support/bugreports// (last access: 16/03/2022).
[4] MathWorks. https://ww2.mathworks.cn/en// (last access: 16/03/2022).
[5] Homa Alemzadeh, Ravishankar K. Iyer, Zbigniew Kalbarczyk, and Jai Raman. 2013. Analysis of Safety-Critical Computer Failures in Medical Devices. *IEEE Security Privacy 11* 4 (2013), 14–26. https://doi.org/10.1109/MSP.2013.49
[6] Olivier Bouissou and Alexandre Chapoutot. 2012. An Operational Semantics for Simulink's Simulation Engine. *ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES 2012)* (2012), 129–138. https://doi.org/10.1145/2248418.2248437
[7] Angelo Brillout, Nannan He, Michele Mazzucchi, Daniel Kroening, Mitra Purandare, Philipp Rümmer, and Georg Weissenbacher. 2009. Mutation-Based Test Case Generation for Simulink Models. *International Symposium on Formal Methods for Components and Objects (FMCO 2009)* (2009), 208–227. https://doi.org/10.1007/978-3-642-17071-3_11
[8] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. 2017. Learning to prioritize test programs for compiler testing. *IEEE/ACM International Conference on Software Engineering (ICSE 2017)* (2017), 700–711. https://doi.org/10.1109/ICSE.2017.70
[9] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An Empirical Comparison of Compiler Testing Techniques. In Proc. International Conference on Software Engineering (ICSE 2016). *ACM* 1 (2016), 180–190. https://doi.org/10.1145/2884781.2884878
[10] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Comput* 53, 1 (2020), 1–36. https://doi.org/10.1145/3363562
[11] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Lu Zhang. 2019. History-Guided Configuration Diversification for Compiler Test-Program Generation. *International Conference on Automated Software Engineering (ASE 2019)* (2019), 305–316. https://doi.org/10.1109/ASE.2019.00037
[12] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2021. Coverage Prediction for Accelerating Compiler Testing. *IEEE Transactions on Software Engineering* 47, 2 (2021), 261–278. https://doi.org/10.1109/TSE.2018.2889771
[13] Shafiul Azam Chowdhury, Soumik Mohian, Sidharth Mehra, Siddhant Gawsane, Taylor T. Johnson, and Christoph Csallner. 2018. Automatically finding bugs in a commercial cyber-physical system development tool chain with SLforge. *International Conference on Software Engineering (ICSE 2018)* (2018), 981–992. https://doi.org/10.1145/3180155.3180231
[14] Shafiul Azam Chowdhury, Sohil Lal Shrestha, Taylor T. Johnson, and Christoph Csallner. 2020. SLEMI:Equivalence Modulo Input (EMI) Based Mutation of CPS Models for Finding Compiler Bugs in Simulink. *International Conference on Software Engineering (ICSE 2020)* (2020), 335–346. https://ieeexplore.ieee.org/abstract/document/9283988
[15] Shafiul Azam Chowdhury, Lina Sera Varghese, Soumik Mohian, Taylor T. Johnson, and Christoph Csallner. 2018. A Curated Corpus of Simulink Models for Model-Based Empirical Studies. *International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS 2018)* (2018), 45–48. https://ieeexplore.ieee.org/abstract/document/8445079
[16] COMBAT. 2022. Replication package of COMBAT. Retrieved Feb 2022 from https://github.com/EDA-Testing/COMBAT
[17] U.S. Consumer Product Safety Commission (CPSC). 2010. Recall 11-702: Fire Alarm Control Panels Recalled by Fire-Lite Alarms Due to Alert Failure. Retrieved Oct.,2010 from http://www.cpsc.gov/en/Recalls/2011/Fire-Alarm-Control-Panels-Recalledby-Fire-Lite-Alarms-Due-to-Alert-Failure
[18] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust Typechecker Using CLP (T). *IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)* (2015), 482–493. https://ieeexplore.ieee.org/document/7372036
[19] Péter Fehér, Tamás Mészáros, László Lengyel, and Pieter J. Mosterman. 2013. Data Type Propagation in Simulink Models with Graph Transformation. *Eastern European Regional Conference on the Engineering of Computer Based Systems (ECBS-EERC 2013)* (2013), 127–137. https://ieeexplore.ieee.org/document/6664519
[20] Christoph Guger, Alois Schlogl, Christa Neuper, Dirk Walterspacher, Thomas Strein, and Gert Pfurtscheller. 2001. Rapid prototyping of an EEG-based braincomputer interface (BCI). *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 9, 1 (2001), 49–58. https://doi.org/10.1109/7333.918276
[21] Grégoire Hamon and John Rushby. 2007. An operational semantics for Stateflow. *International Journal on Software Tools for Technology Transfer* 9, 5 (2007), 447–456. https://doi.org/10.1007/s10009-007-0049-7

[22] MathWorks Inc. 2018. Products and Services. Retrieved Jan,2020 from http://www.mathworks.com/products/

[23] MathWorks Inc. 2022. Simulink Documentation — MATLAB Simulink. Retrieved Feb 2022 from http://www.mathworks.com/help/simulink/

[24] He Jiang, Zhide Zhou, Zhilei Ren, Jingxuan Zhang, and Xiaochen Li. 2021. CTOS: Compiler Testing for Optimization Sequences of LLVM. *IEEE Transactions on Software Engineering* (2021), 1–1. https://doi.org/10.1109/TSE.2021.3058671

[25] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs. *ACM Sigplan Notices* 49, 6 (2014), 216–226. https://doi.org/10.1145/2666356.2594334

[26] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. *ACM SIGPLAN Notices* 50, 10 (2015), 386–399. https://doi.org/10.1145/2858965.2814319

[27] Edward A. Lee. 2008. Cyber Physical Systems: Design Challenges. *International Symposium on Object Oriented Real-Time Distributed Computing (ISORC 2008)* (2008), 363–369. https://doi.org/10.1109/ISORC.2008.25

[28] Edward A. Lee. 2010. Cyber Physical Systems: Design Challenges. *CPS foundations. In Proc. Design Automation Conference (DAC 2010)* (2010), 737–742. https://doi.org/10.1145/1837274.1837462

[29] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastari F. Donaldson. 2015. Many-core compiler fuzzing. *ACM SIGPLAN Notices* 50, 6 (2015), 65–76. https://doi.org/10.1145/2813885.2737986

[30] Bing Liu, Lucia, Shiva Nejati, and Lionel C. Briand. 2017. Improving fault localization for Simulink models using search-based testing and prediction models. *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER 2017)* (2017), 359–370. https://doi.org/10.1109/SANER.2017.7884636

[31] William M. McKeeman. 1998. Differential testing for software. *Digital Technical Journa* 10, 1 (1998), 100–107.

[32] Pieter J. Mosterman, Justyna Zander, Grégoire Hamon, and Ben Denckla. 2009. Towards computational hybrid system semantics for time-based block diagrams. *IFAC Conference on Analysis and Design of Hybrid Systems (ADHS 2009)* (2009), 376–385. https://doi.org/10.3182/20090916-3-ES-3003.00065

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

[34] Marta Olszewska, Yanja Dajsuren, Harald Altinger, Alexander Serebrenik, Marina A.Waldén, and Mark G. J. van den Brand. 2016. Tailoring complexity metrics for Simulink models. *In Proc. 10th European Conference on Software Architecture Workshops.* (2016), 5–5. https://dl.acm.org/doi/10.1145/2993412.3004853

[35] Vera Pantelic, Steven Postma, Mark Lawford, Monika Jaskolka, Bennett Mackenzie, Alexandre Korobkine, Marc Bender, Jeff Ong, Gordon Marks, and Alan Wassyng. 2017. Software engineering practices and Simulink: Bridging the gap. *International Journal on Software Tools for Technology Transfer* 20, 1 (2017), 95–117. https://doi.org/10.1007/s10009-017-0450-9

[36] Akshay Rajhans, Srinath Avadhanula, Alongkrit Chutinan, Pieter J. Mosterman, and Fu Zhang. 2018. Graphical modeling of hybrid dynamics with Simulink and Stateflow. *International Conference on Hybrid Systems: Computation and Control (HSCC 2018)* (2018), 247–252. https://doi.org/10.1145/3178126.3178152

[37] Steven Rasmussen, Jason Mitchell, Chris Schulz, Corey Schumacher, and Phillip Chandler. 2012. A multiple UAV simulation for researchers. *In AIAA Modeling and Simulation Technologies Conference and Exhibit.* (2012), 11–14. https://doi.org/10.2514/6.2003-5684

[38] Prahladavaradan Sampath, A. C. Rajeev, S. Ramesh, and K. C. Shashidhar. 2007. Testing Model-Processing Tools for Embedded Systems. *IEEE Real- Time and Embedded Technology and Applications Symposium* (2007), 209–214. https://doi.org/10.1109/RTAS.2007.39

[39] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A comprehensive study of deep learning compiler bugs. *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)* (2021), 968–980. https://doi.org/10.1145/3468264.3468591

[40] Sohil Lal Shrestha, Shafiul Azam Chowdhury, and Christoph Csallner. 2020. DeepFuzzSL: Generating models with deep learning to find bugs in the Simulink toolchain. *Workshop on Testing for Deep Learning and Deep Learning for Testing (DeepTest 2020)* (2020). https://par.nsf.gov/servlets/purl/10187922

[41] Sohil Lal Shrestha and Christoph Csallner. 2021. SLGPT: Using Transfer Learning to Directly Generate Simulink Model Files and Find Bugs in the Simulink Toolchain. *Evaluation and Assessment in Software Engineering(EASE 2021)* (2021), 260–265. https://doi.org/10.1145/3463274.3463806

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

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

[44] Chengnian Sun, Vu Le, and Zhengdong Su. 2016. Finding Compiler Bugs via Live Code Mutation. *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)* (2016), 849–863. https://doi.org/10.1145/2983990.2984038

[45] Yixuan Tang, He Jiang, Zhide Zhou, Xiaochen Li, Zhilei Ren, and Weiqiang Kong. 2021. Detecting Compiler Warning Defects Via Diversity-Guided Program Mutation. *IEEE Transactions on Software Engineering* (2021), 1–1. https://doi.org/10.1109/TSE.2021.3119186

[46] Yixuan Tang, Zhilei Ren, Weiqiang Kong, and He Jiang. 2020. Compiler testing: a systematic literature analysis. Frontiers Comput. *Sci* 14, 1 (2020), 14–26. hhttps://doi.org//10.1007/s11704-019-8231-0

[47] Sandro Tolksdorf, Daniel Lehmann, and Michael Pradel. 2019. Interactive metamorphic testing of debuggers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 273–283.

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

[49] Yibiao Yang, Yuming Zhou, Hao Sun, Zhendong Su, Zhiqiang Zuo, Lei Xu, and Baowen Xu. 2019. Hunting for Bugs in Code Coverage Tools via Randomized Differential Testing. *International Conference on Software Engineering (ICSE 2019)* (2019), 488–499. https://doi.org/10.1109/ICSE.2019.00061

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

[51] Zhide Zhou Zhilei Ren, Guojun Gao and He Jiang. 2021. An empirical study of optimization bugs in GCC and LLVM. *J. Syst. Softw* 174, 110884 (2021). https://doi.org/10.1016/j.jss.2020.110884