Partition Based Differential Testing for Finding Embedded Code Generation Bugs in Simulink

He Jiang[a], Hongyi Cheng[b], Shikai Guo[b], Xiaochen Li[a]
[a]School of Software, Dalian University of Technology, Dalian, China
[b]School of Information Science and Technology, Dalian Maritime University, Dalian, China
jianghe@mail.dlut.edu.cn, chenghongyi@dlmu.edu.cn, shikai.guo@dlmu.edu.cn, xiaochen.li@dlut.edu.cn

2023 60th ACM/IEEE Design Automation Conference (DAC) | 979-8-3503-2348-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/DAC56929.2023.10247877

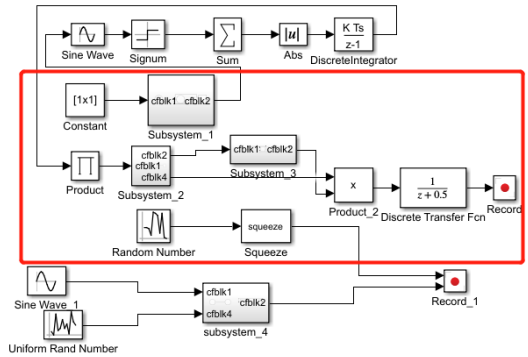
***Abstract*—Engineers frequently generate embedded code from Simulink models for control applications. However, target applications using the code could behave unexpectedly, due to the bugs in code generation. In this study, we propose MOPART, the first model partition based differential testing method for code generation testing in Simulink. MOPART uses multiple-way network partitioning to generate diverse bug-triggering Simulink models to thoroughly exercise the code generation process. MOPART then finds bugs by analyzing the outputs of these Simulink models with differential testing. Experiments show that MOPART significantly outperforms existing approaches, which finds 11 confirmed code generation bugs in only two weeks.**

***Index Terms*—Code generation, Simulink coder, differential testing, Simulink**

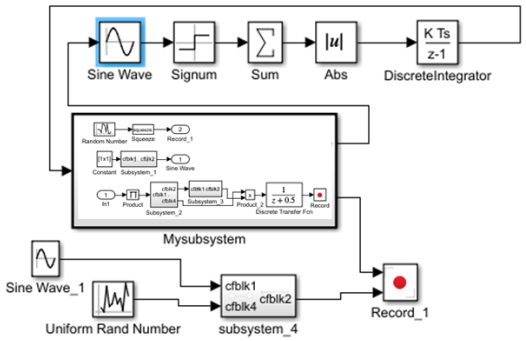
I. Introduction

Simulink is a fundamental platform for engineers to design and analyze models for control applications [1]. When the designed Simulink model satisfies engineers' requirements, engineers can transform the Simulink model into embedded code (e.g., C source code), which can get deployed in safety-critical applications such as aerospace and healthcare. However, the code generation process of Simulink could have bugs, due to the wrong or improper implementation of this process. These bugs may inject unexpected behaviors to target applications via the generated code, causing unexpectedly catastrophe consequences for safety-critical applications.

An example of this type of bug is shown in Fig. 1. We have two Simulink models A and B. Simulink model B is generated by setting blocks in the red box in Simulink model A as a subsystem. In practice, after designing a Simulink model, engineers usually refactor the Simulink model to put blocks implementing similar functions as subsystems. This kind of refactoring improves the maintainability of the Simulink model; it does not change its functionality. Hence, the two Simulink models have the same simulation results in the Normal mode simulation[1]. However, when we execute the generated code, we get different results. This bug has been confirmed by Simulink engineers (with bug ID#05794231). The cause of this bug is that during code generation, Simulink transforms the *Sine Wave* block in blue in Fig. 1(B) into the 'sine' method in the Standard C Library. The 'sine' method works normally with inputs between $-\pi$ and π;



(a) Simulink model A



(b) Simulink model B which is equivalent with A

Fig. 1: Example of a code generation bugs

but for extremely large inputs (e.g., 1.74E+17), this method transforms the input value with the 'eps' method, and takes the transformed value as input, leading to a different output. If such generated code is deployed on safety-critical applications, in extreme cases, we could get unexpected and weird behaviors on the applications. Even worse, engineers may not be aware of this difference, leading potential safety hazard.

Therefore, finding bugs in Simulink code generation is important. A typical approach to find such bugs is Software-in-the-Loop (SIL) simulation[2]. Given a Simulink model (which are usually randomly generated), we generate embedded code from the Simulink model. SIL encapsulates the generated source code as a block using the *s-function* block in Simulink.

[1]When designing Simulink models, engineers usually execute them in the simulation mode with Simulink language specifications. After designing, engineers generate embedded code based on target language specifications.

[2]SIL and PIL Simulations. https://www.mathworks.com/help/ecoder/ug/about-sil-and-pil-simulations.html

It integrates this block into a Simulink model. SIL executes the original Simulink model and the integrated Simulink model with certain inputs, and compares their outputs to find unexpected outputs after code generation.

However, existing approaches may not find code generation bugs effectively due to the challenges in constructing bug-triggering Simulink models and deciding test oracles. On the one hand, Simulink is commercial software implemented by professional engineers. A randomly constructed Simulink model may not trigger any Simulink bugs. It is a challenge to construct diverse and complex Simulink models that can trigger code generation bugs. On the other hand, when generating code from Simulink models, the test oracle is hard to design (i.e., the expected code to generate). It may be not sufficiently to simply compare the outputs before and after code generation. Hence, it is a challenge to decide test oracles for testing the generated code.

In this paper, we propose MOPART, a model partition based differential testing method for code generation testing. MOPART aims to generate diverse and complex Simulink models to exercise the code generation process thoroughly for finding bugs. Given a randomly generated Simulink model, MOPART uses multiple-way network partitioning to partition blocks in the Simulink model into groups. MOPART generates variant Simulink models by randomly setting blocks in different groups in the seed Simulink model as subsystems. Since these subsystems increase the complexity of Simulink models, they may trigger more bugs during code generation. To address the test oracle challenge, instead of analyzing the outputs of a Simulink model before and after code generation, MOPART takes the output of the seed Simulink model as oracle, and compares this output with the variant Simulink models through differential testing. Since subsystems should not affect the functionality of Simulink models, the seed Simulink model and its variants are equivalent. If MOPART finds output inconsistency of generated code from these Simulink models, a code generation bugs could be triggered.

Experiments show that, within the same testing period, MOPART significantly outperforms existing approaches in finding output inconsistency of the generated code in Simulink. MOPART is effective in finding bugs, which finds 11 code generation bugs in the latest Simulink version (i.e, Simulink 2022a) in only two weeks. Simulink may improperly deal with certain C language specifications, subsystems, and special input values during code generation, possibly leading unexpected behaviors of target applications.

The main contributions of this work are:

- To the best of our knowledge, MOPART is the first study to systematically find code generation bugs in Simulink with partition-based differential testing.
- Extensive experiments are conducted to assess the bug-finding capability of MOPART. MOPART significantly outperforms existing approaches.
- We release MOPART as a replication package for code generation testing [2].

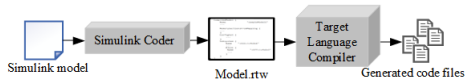


Fig. 2: The process of code generation in Simulink

II. Related Work

Our work is related to Simulink testing. Since Simulink is widely used for designing embedded systems [1], the correctness of Simulink is important for design automation.

Most of the work find Simulink bugs using randomly generated Simulink models. If Simulink crashes when executing a model, a bug could be triggered. Chowdhury et al. [3] propose CyFuzz. CyFuzz randomly generates an initial Simulink model and iteratively fixes its grammar errors. Another approach is SLforge [4], which uses the language specification of Simulink to randomly generate Simulink models. In addition, some learning based approaches are proposed [5], [6] to learn the relationship of block connections with deep learning or transfer learning for Simulink model generation. Among these approaches, SLfroge is the state-of-the-art Simulink model generator. However, a randomly generated Simulink model may not trigger Simulink bugs effectively.

Another line of approaches for Simulink testing is based on differential testing. They systematically mutate a seed Simulink model as long as its semantics remain equivalent under a given input [7]. The seed Simulink model and its equivalent variants are used to find Simulink bugs if the compilation of the two models produces different results. However, these approaches only focus on testing the compilation system of Simulink. No source code are generated during the testing.

Different from existing studies, we aim to find code generation bugs in Simulink. We propose MOPART to thoroughly test the code generation process. Although several approaches aim to optimize embedded code generation of Simulink [1], [8], to the best of our knowledge, this is the first approach to systematically find code generation bugs.

III. Background of Code Generation in Simulink

Simulink is widely used in an industrial context for engineers to design, simulate, and generate embedded code for control applications [9]. Simulink provides a block diagram environment for model-based design with Simulink models. As shown on the left top corner of Fig. 1, a Simulink model is a set of blocks that are connected by signals specifying the flow of data. Each block receives data from its input ports, performs some operations, and passes the results to its subsequent blocks through the output ports and connection lines. A block in Simulink has properties such as block type (e.g., a "*If*" block) and datatype (e.g., double, unit32). Each block can have child blocks to form a hierarchical subsystem structure (via Simulink's Subsystem and Model Referencing).

To deploy the designed Simulink model on a target application, Simulink has an important feature to generate embedded code from Simulink models [1]. As shown in Fig. 2, the component Simulink Coder automatically transforms a

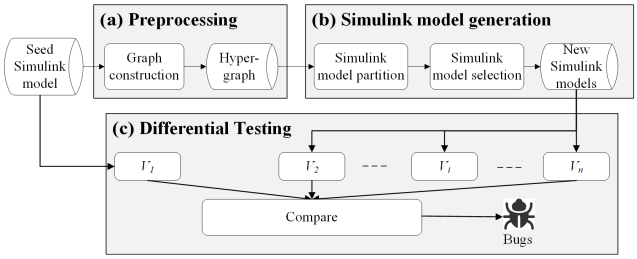


Fig. 3: The overview of MOPART

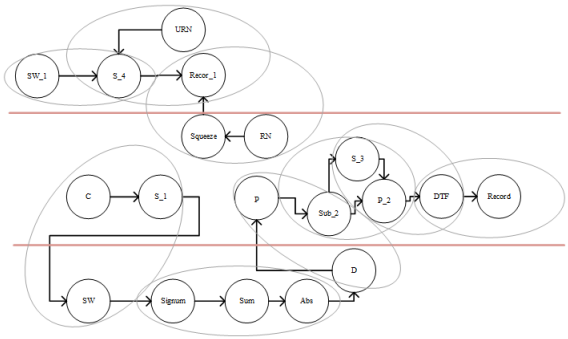


Fig. 4: Hypergraph for the Simulink model in Fig. 1(a)

Simulink model into a model.rtw file, which includes the model-specific information required for code generation, such as the target language (e.g., C/C++) and the code optimization level. Simulink Coder can also specify the atomic unit of a Simulink model. Simulink blocks related to an atomic unit usually implement the same functionality, which are organized in the same generated code file for reuse. This process only modifies the structure of generated code. The functionality of the generated code remains the same. Simulink Coder then passes all the information to the Target Language Compiler (TLC), which uses the transformed information in combination with a set of included system target files and block target files to generate the code. The generated code can be used for real-time and nonreal-time applications.

IV. MOPART Framework

To effectively find code generation bugs in Simulink, we propose MOPART, a **MO**del **PART**ition based differential testing approach for code generation testing. The framework of MOPART is illustrated in Fig. 3. MOPART consists of three components: preprocessing, Simulink model generation, and differential testing. The basic idea of MOPART is to generate diverse and complex Simulink models to exercise the code generation process thoroughly for bug finding. The input of MOPART is a seed Simulink model, which can be generated automatically. MOPART preprocesses the seed Simulink model to get its basic information (e.g., the number of blocks). To generate diverse and complex Simulink models, MOPART transforms the seed Simulink model as a hypergraph. In Simulink model generation, MOPART performs model partitioning on the hypergraph. Each partition contains a set of blocks that are strongly connected with each other. MOPART randomly selects some partitions, and sets blocks in each partition as a subsystem. The output of this step is a set of new Simulink models with different subsystem structures. Since these new Simulink models have more complex structures, they may better trigger code generation bugs, thus addressing the bug-triggering Simulink model construction challenge. To address the test oracle challenge, MOPART conducts differential testing on these Simulink models. MOPART sets each new subsystem as an atomic unit for code generation. Since the only difference of these Simulink models is the structure of the generated source code (i.e., blocks transformed as a single source code file), these Simulink models are equivalent to provide the same outputs. When the outputs of them are different under certain inputs, code generation bugs could be triggered.

A. Preprocessing

In this study, we find code generation bugs by generating new Simulink models. To this end, we analyze the structure of the seed Simulink model. Since Simulink models are block-diagrams, in the preprocessing MOPART transforms a Simulink model as a graph to further analysis. Specifically, given a seed Simulink model, MOPART first gets the basic information of the model (such as the number of blocks and connections, the type of blocks). Based on this information, a hypergraph is constructed from the seed Simulink model. where each vertex in the hypergraph represents a block in the Simulink model, and the edges are the connections between different blocks. In a hypergraph, an edge is a hyperedge, which can connect more than two vertices. Since a signal in a Simulink model can pass to more than one block simultaneously, in the hypergraph, we use the hyperedge to connect blocks which take the same signal as input or output to reflect this characteristic. For example, Fig. 4 is the hypergraph of the Simulink model in Fig. 1(A). We configure the hyperedge to connect at most three blocks. Each circle in the figure is a hyperedge. The hyperedge on the top connects S_4 (*Subsystem_4*), URN (*Uniform Rand Number*), Recor_1 (*Record_1*) blocks. The output of this step is the hypergraph representing the seed Simulink model.

B. Simulink Model Generation

This step generates new Simulink models from the seed Simulink model. The intuition of Simulink model generation is that blocks frequently connected with each other in a Simulink model tend to implement similar functions of the target application. By setting a group of blocks as a subsystem, new Simulink models can be generated. Since we only modify the structure of Simulink models, these Simulink models are equivalent in terms of the functionality. However, by creating subsystems, the complexity and the diversity of Simulink models could be improved. Such Simulink models can thoroughly test the code generation process of Simulink, thus having a higher chance to trigger code generation bugs.

Algorithm 1 Simulink Model Partition:

Input: A hypergraph H representing the seed Simulink model, the number of partitions n_p
Output: The target partition of the seed Simulink model
1: $P = startingPartition(H, n_p)$;
2: **repeat**
3: Information I, Cost $C = initPartition(P)$;
4: New partition $P = newPartition(P, n_p, I, C)$;
5: **until** there is no improvement in C.

Simulink model partition: In this study, we identify blocks that can be put into a subsystem as a graph partition problem, which aims to partition a graph into sub-graphs based on a cost function. We partition the hypergraph representing the seed Simulink model with the Fiduccia-Mattheyses (FM) algorithm [10], a classical hypergraph partition algorithm.

The main process of Simulink model partition is presented in Algorithm 1. The inputs of this algorithm are the hypergraph representing the seed Simulink model and a parameter n_p indicating the number of partitions to be created. The output is the target partition of the seed Simulink model. The basic assumption of FM algorithm is that it is better to put all blocks in the same hyperedge to the same partition.

MOPART first randomly creates n_p partitions, and puts each block into a partition. Initially, the algorithm sets all the blocks as unlocked, which can be moved from one partition to another. MOPART then calls $initPartition()$ the get the statistical information of P. The information includes the number of blocks and hyperedges in each partition, and the cost of this partition. The cost is defined as the number of hyperdedges across two partitions.

Based on this information, MOPART creates new partitions for the Simulink model with the method $newPartition()$ to reduce the cost that can be obtained. The algorithm tries to move every unlocked block from the current partition to a different one; it computes the cost (i.e., the number of cross-partition hyperedges) can be reduced for each move. The algorithm chooses one of the best block to move, and generates new partitions. Meanwhile, it marks this block as unlocked.

This process repeats until the cost cannot be reduced for all new partitions. The red lines in Fig. 4 is an example to divide a hypergraph into three partitions.

Partition selection: According to the partition of the seed Simulink model, MOPART creates subsystems based on a parameter n_s (i.e., the number of new subsystems to be created). MOPART randomly selects n_s partitions in the seed Simulink model. For each partition, MOPART creates a subsystem by adding all the blocks in the partition to a parent block. In this way, new Simulink models are generated. These Simulink models implement the same functionality as the seed Simulink models. We can repeat the partition selection process to generate a set of Simulink models.

Simulink models generated by MOPART could introduce algebraic loops, i.e., a circular data dependence path on which all blocks are direct feed-through. In an algebraic loop, Simulink would require the output of a block for computing the block's input [7]. Although Simulink can solve some algebraic loops, it is computationally expensive. Since MOPART can efficiently generate a large number of Simulink models by selecting different partitions, MOPART directly discards new Simulink models with algebraic loops in this study.

The outputs of this step are the new Simulink models generated from the seed Simulink model.

C. Differential testing

We find code generation bugs in Simulink with differential testing [11]. The intuition of differential testing is to run different but equivalent test cases (e.g., Simulink models) in the same environment. The equivalence means that although two Simulink models are different in structure, they are expected to get the same outputs with the same inputs and configurations. If the outputs are different, a bug in executing these Simulink models could be triggered.

In the literature, a possible way to find code generation bugs is SIL simulation. As explained in Section I, SIL compares the output of a randomly generated Simulink model before and after code generation to find inconsistent outputs. However, this approach may be ineffective, since it highly relies on the quality of the randomly generated Simulink model.

In this study, MOPART uses the newly generated Simulink models for differential testing. MOPART sets each newly created subsystem in a new Simulink model as an atomic unit with Simulink Coder. By creating atomic units, MOPART increases the complexity and the frequency of data exchange of Simulink models during code generation, which could thoroughly test the code generation process. As we mentioned in Section III, atomic units only modify the structure of the generated source code. The new Simulink models are expected to generate equivalent source code with the seed Simulink model. Given an input, if the seed Simulink model and a new Simulink model get the same outputs in the Normal simulation mode of Simulink, but the generated code of them provide different outputs, a code generation bugs in Simulink could be triggered. In this study, two output values v_1 and v_2 are different (or inconsistent) if the relative difference between them (i.e., $\frac{|v_1-v_2|}{\min\{v_1,v_2\}}$) is higher than a threshold. As suggested by Simulink[3], we set the threshold as 1E-3.

V. Evaluation

In this section, three research questions (RQs) are investigated to evaluate the effectiveness of MOPART. Specifically, RQ1 compares the bug finding capability of MOPART with existing approaches. RQ2 investigates the impact of the number of partitions on MOPART. RQ3 analyze whether MOPART can find real code generation bugs in Simulink.

A. Evaluation Setup

Seed Simulink models: MOPART takes seed Simulink models as inputs. In the evaluation, we use the widely used Simulink model generator SLforge [4] to generate seed Simulink models. SLforge supports to randomly generate

[3]https://www.mathworks.com/help/simulink/gui/relative-tolerance.html

TABLE I: Effectiveness of MOPART and baselines

	Default	**Random**	**MOPART**
# of seed Simulink models	1,248	1,248	888
# of new Simulink models	NA	288	538
success rate	NA	0.23	0.61
# of output inconsistency	60	21	102

TABLE II: Results of MOPART with different n_p values

	n_p=5	n_p=10	n_p=15	n_p=20
# of seed Simulink models	807	679	802	628
# of new Simulink models	254	635	797	625
success rate	0.31	0.94	0.99	0.99
# of output inconsistency	23	56	83	53

Simulink models with most Simulink features. According to the suggestion of SLforge and our computation resources, the depth (i.e., hierarchy) of each seed Simulink model is at most three. There are 100 blocks in each seed Simulink model.

Implementation: MOPART is implemented in MATLAB. We set the number of partitions n_p to be created on the seed Simulink model as 15. During Simulink model generation, for each seed Simulink model, we randomly select one partition as subsystem to generate a new Simulink model. 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.

B. RQ1: Effectiveness of MOPART compared with baselines

We compare the effectiveness of MOPART with two baselines, i.e., `Default` and `Random`. `Default` finds code generation bugs in Simulink only with seed Simulink models. `Default` follows SIL to compare the outputs of the seed Simulink model before and after code generation to find output inconsistency. `Random` generates new Simulink models by randomly selecting a subset of blocks in the seed Simulink model as a subsystem, instead of conducting Simulink model partitions. It then uses the same differential testing process as that in Section IV-C to find output inconsistency.

We execute MOPART and the baselines on the latest version of Simulink (i.e., Simulink R2022a) for two weeks, which includes the time for generating seed Simulink models, generating new Simulink models (if necessary), and testing. We compare the number of executed seed Simulink models, the success rate of generating new Simulink models, and the number of output inconsistency detected.

As shown in Table I, MOPART, `Default`, and `Random` process 888, 1,248, and 1,248 seed Simulink models in two weeks, respectively. MOPART is relatively slow because it requires additional time to conduct Simulink model partition. Since both MOPART and `Random` generate new Simulink models, we compare the success rate of this process. `Random` generates a large number of invalid Simulink models, because randomly creating subsystems could break the language specification of Simulink. In contrast, MOPART successfully generates 538 new Simulink models. The success rate is 60.6%. Regarding the ability in finding code generation inconsistency, MOPART finds 102 inconsistencies between the seed Simulink model and the new Simulink models, which number is significantly higher than that of `Default` and `Random`.

We remark that in this RQ, some output inconsistencies could be caused by the same reason. We take the number of output inconsistency as a proxy of the effectiveness of an approach in finding code generation bugs, since a high number of output inconsistency usually indicates a higher probability in finding real code generation bugs. We do not manually investigate the root causes of these output inconsistencies, due to the high cost to manually analyze the causes. We conduct a preliminary analysis on the output inconsistency in Section V-D with Simulink engineers to show the effectiveness of MOPART in finding real code generation bugs.

Conclusion: MOPART is effective in generating Simulink models to trigger more code generation inconsistency compared with baselines.

C. RQ2: Impact of the number of partitions

To assess the impact of the number of partitions n_p, we set n_p from 5 to 20 with a step size of 5. We run MOPART with each value of n_p, and compute the same metrics as those used in RQ1.

As shown in Table II, n_p affects the efficiency and effectiveness in finding output inconsistency. MOPART processes between 628 and 807 seed Simulink models in two weeks with different n_p values. The number of processed seed Simulink models depends on the complexity of partitioning a hypergraph into a certain number of partitions. When we increase the number of partitions, the success rate in generating new Simulink models increases. For example, MOPART successfully generates 254 new Simulink models from seed Simulink models with 5 partitions with success rate of 0.31. With 20 partitions, the success rate increases to 0.99. The success rate affects the number of Simulink models used for testing. When the success rate is low, MOPART spends most of the time in generating valid Simulink models. Hence, it cannot effectively find code generation bugs. When $n_p = 5$, MOPART finds 23 output inconsistencies. In contrast, when generating new Simulink models based on 15 partitions, MOPART finds 83 output inconsistencies. With a high number of output inconsistency, MOPART has a higher probability to find real code generation bugs. However, a very large number of n_p may not be useful. For example, when n_p is 20, given a seed Simulink model with 100 blocks, there are only 5 blocks on average in each partition. When creating subsystems on a small number of blocks, such simple subsystems may not be effective in triggering different types of code generation bugs. MOPART only finds 53 output inconsistencies.

Conclusion: The number of partitions affects the efficiency and effectiveness in generating Simulink models and finding output inconsistency. MOPART can find a high number of output inconsistency with $n_p = 15$.

D. RQ3: Effectiveness in finding real code generation bugs

This RQ assesses the effectiveness of MOPART in finding real code generation bugs in Simulink. We analyze the root

TABLE III: The detail of bugs found by MOPART

#	ID	Title
1	05794231	Incorrect code generation
2	05807041	Code generation error caused by subsystem
3	05824672	SIL exception caused by Complex to Real-Img in subsystem
4	05824768	The Unary Minus module loses data after generating codes
5	05829052	The delay module in a multilayer subsystem raises an error
6	05842603	The MinMax Runing Resettable module lost data
7	05842655	Model code exception caused by bias module in subsystem
8	05842869	Repeating Sequence Interpolated Generating error signals
9	05862281	The Polyval module has abnormal data after code generation
10	05862510	Math function error handling in code generation
11	05864689	Trigonometric Function module incorrectly used

causes of the output inconsistency found by MOPART in RQ1. For each inconsistency, we compare the failed assertion and back-trace of this output inconsistency with previously found output inconsistency to detect duplicate results. We consider each non-duplicate results as a bug. For each non-duplicate bug, we conduct Simulink model reduction. We remove blocks in the corresponding Simulink model sequentially, and check whether the bug can still be triggered after the block removal. At last, a minimum set of blocks that can trigger this bug is found. We report the reduced bug and the corresponding Simulink model to Simulink engineers for confirmation.

Currently, 11 bugs have been confirmed by engineers (as shown in Table III). These bugs are related to various blocks, such as *Delay*, *Bias*, *Polyval*, and *Math* blocks. We release the Simulink models that trigger bugs on GitHub [2].

First, the difference between language specifications of Simulink and C/C++ can cause code generation bugs. An example is bug ID#05794231 explained in Fig. 1.

Second, Simulink may improperly deal with subsystems during code generation. In bug ID#05824768, MOPART generates a Simulink model with multiple-level of subsystems. When transforming this Simulink model into C code, the data passing through the *Unary Minus* block is lost, leading to output inconsistency. A similar bug is bug ID#05824672.

Third, the generated code may not correctly compute special input values. For bug ID#05842603, MOPART generates a Simulink model containing a *MinMax Runing Resettable* block. The input of this block is 'NaN'. When transforming this Simulink model to C code, Simulink did not provide suitable mechanism to handle 'NaN' for this block, causing an output inconsistency. However, there is no warning to alarm users to check the special input value 'NaN'.

RQ3 shows that MOPART is useful in finding code generation bugs automatically in practice. Although the automatically generated Simulink models may be different from those designed by engineers, these bugs are still threads to real developing scenarios. We find some bugs are caused by the difference between Simulink specification and C/C++ language specifications. However, there should be warning or addition checks to help engineers aware of such differences during the design phase, rather than injecting unexpected behaviors to target applications.

Conclusion: MOPART can find real code generation bugs related to various Simulink blocks. MOPART find 11 confirmed code generation bugs in Simulink.

VI. Conclusion

This study proposes MOPART to find code generation bugs in Simulink, which is important to improve the reliability of design automation. MOPART addresses two main challenges in constructing bug-triggering Simulink models and deciding test oracles. Given a seed Simulink model, MOPART partitions the model based on its hypergraph. MOPART generates new Simulink models by creating subsystems from different partitions. At last, code generation bugs are found by executing the seed Simulink model and new Simulink models with differential testing. Within only two weeks, we find 11 confirmed bugs on the latest version of Simulink (i.e., Simulink R2022a). Compared with baselines, MOPART can find significantly more output inconsistencies between Simulink models.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (No.62032004, No.62202079), the Dalian Excellent Young Project No.2022RY35, and the Fundamental Research Funds for the Central Universities No.DUT22RC(3)028.

References

[1] Zhuo Su, Zehong Yu, Dongyan Wang, Yixiao Yang, Yu Jiang, Rui Wang, Wanli Chang, and Jiaguang Sun. HCG: optimizing embedded code generation of simulink with SIMD instruction synthesis. In *Proc. of ACM/IEEE Design Automation Conf. (DAC)*, pages 1033–1038, 2022.

[2] Code and bugs. https://github.com/Simulink-Testing-Code/C2C-Testing.

[3] Shafiul Azam Chowdhury, Taylor T. Johnson, and Christoph Csallner. Cyfuzz: A differential testing framework for cyber-physical systems development environments. In *Cyber Physical Systems Design, Modeling, and Evaluation*, pages 46–60, Cham, 2017. Springer Int'l Publishing.

[4] Shafiul Azam Chowdhury, Soumik Mohian, Sidharth Mehra, Siddhant Gawsane, Taylor T Johnson, and Christoph Csallner. Automatically finding bugs in a commercial cyber-physical system development tool chain with slforge. In *Proc. of Int'l Conf. on Software Eng. (ICSE)*, pages 981–992, 2018.

[5] Sohil Lal Shrestha, Shafiul Azam Chowdhury, and Christoph Csallner. Deepfuzzsl: Generating models with deep learning to find bugs in the simulink toolchain. In *Workshop on Testing for Deep Learning and Deep Learning for Testing (DeepTest)*, 2020.

[6] Sohil Lal Shrestha and Christoph Csallner. Slgpt: using transfer learning to directly generate simulink model files and find bugs in the simulink toolchain. In *Evaluation and Assessment in Software Engineering*, pages 260–265. 2021.

[7] Shafiul Azam Chowdhury, Sohil Lal Shrestha, Taylor T Johnson, and Christoph Csallner. SLEMI: Equivalence modulo input (EMI) based mutation of cps models for finding compiler bugs in simulink. In *Proc. of Int'l Conf. on Software Eng. (ICSE)*, pages 335–346. IEEE, 2020.

[8] Zhuo Su, Dongyan Wang, Yixiao Yang, Yu Jiang, Wanli Chang, Liming Fang, Wen Li, and Jiaguang Sun. Code synthesis for dataflow-based embedded software design. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 41(1):49–61, 2021.

[9] Nannan He, Philipp Rümmer, and Daniel Kroening. Test-case generation for embedded simulink via formal concept analysis. In *Proc. of Design Automation Conf. (DAC)*, pages 224–229, 2011.

[10] Charles M Fiduccia and Robert M Mattheyses. A linear-time heuristic for improving network partitions. In *Papers on Twenty-five years of electronic design automation*, pages 241–247. 1988.

[11] Dongning Ma, Jianmin Guo, Yu Jiang, and Xun Jiao. Hdtest: Differential fuzz testing of brain-inspired hyperdimensional computing. In *Proc. of Design Automation Conf. (DAC)*, pages 391–396. IEEE, 2021.