CFTCG: Test Case Generation for Simulink Model through Code Based Fuzzing

Zhuo Su[†], Zehong Yu[†], Dongyan Wang[‡], Rui Wang[§], Yang Tao[¶], Yu Jiang[✉ †]

[†] KLISS, BNRist, School of Software, Tsinghua University, Beijing, China

[‡] Information Technology Center, Renmin University of China, Beijing, China

[§] Information Engineering College, Capital Normal University, Beijing, China

[¶] HUAWEI Technologies Co. LTD., Shanghai, China

ABSTRACT

Simulink is extensively utilized in system design for its ability to facilitate modeling and synthesis of embedded controllers. It provides automatic test case generation to assist testers in inspecting the model. However, with the continuous increase in the model's scale, the control logic and internal states of the model are becoming more and more complex. Mainstream test case generation methods based on constraint solving and model simulation face challenges in achieving high coverage metrics.

In this paper, we propose CFTCG, a fuzzing based test case generation method for Simulink models. First, CFTCG generates the fuzzing code, which includes the fuzz driver based on the model's input information and the fuzz code with model-level branch instrumentation. These codes are then compiled together to execute the model oriented fuzzing loop. During this fuzzing loop, we make use of the field information of the model inports and the coverage difference between iterative executions, allowing for more targeted input mutation. We evaluated CFTCG on several benchmark Simulink models. In comparison to the built-in Simulink Design Verifier and the state-of-the-art academic work SimCoTest, CFTCG demonstrates an average improvement of 47.2% and 100.8% on Decision Coverage, 38.3% and 44.6% on Condition Coverage, and 144.5% and 232.4% on Modified Condition Decision Coverage, respectively.

KEYWORDS

Test Case Generation, Simulink Model, Fuzzing, Code Generation

1 INTRODUCTION

Simulink [13] is widely recognized as one of the most popular model-driven design tools, particularly in the context of embedded systems [11, 15]. It supports efficient modeling, fast simulation, and high-quality code generation for embedded control models [8, 16]. To ensure the security and stability of models, it is crucial to conduct thorough testing [4]. However, manual construction of test cases is not only labor-intensive but also falls short of comprehensively testing all elements of the model. Automatic test case generation has proven to be beneficial as it saves significant effort and covers logic that is challenging to detect manually [2].

Currently, there is a substantial body of research focused on test case generation for Simulink models [1, 6, 10, 12]. These works can generally be classified into two types. The first is based on the constraint solving method, exemplified by the Simulink built-in toolkit Simulink Design Verifier (SLDV) [5]. This approach involves transforming the model into a specific formal representation and employing a formal solver to address constraints related to various branch logic in the model. Ultimately, it generates model inputs that satisfy all the imposed constraints. The second is based on the model simulation method, represented by SimCoTest [9]. This approach uses some well-designed random strategies to generate input data for the model and extracts feedback coverage information by executing the model. This feedback information is then used to further optimize the process of test case generation.

Yu Jiang is the corresponding author.

Despite the significant progress achieved by the aforementioned existing works in Simulink model testing, generating high-coverage test cases for models that feature complex control logic remains a challenging task. These models often encompass intricate computing logic and diverse internal states, posing considerable challenges for existing constraint solving based methods to formalized solving. Notably, state space explosion remains a persistently challenging and intractable problem for formal methods. On the other hand, although the model simulation methods have been extensively explored to improve the speed of test case generation by utilizing model information, the dynamic simulation of the model during the test case generation process has a huge impact on the efficiency of test case output. This is mainly due to the simulation process relying heavily on model interpretation, leading to extensive calculations by the simulation engine, which significantly reduces the efficiency of test case generation.

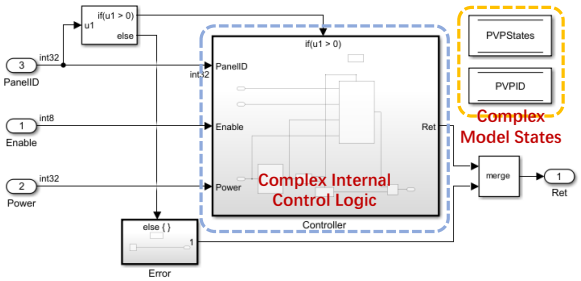


Figure 1: An example model with complex control logic. This is a Solar PV Panel Energy Output Control System. It contains a lot of control logic and states about PV panels.

Figure 1 shows a real industrial model representing a solar PV panel energy output control system. This system interfaces with multiple solar PV panels concurrently and it adjusts the method of electrical energy storage based on the electrical energy output power of the PV panels. The model encompasses intricate electrical energy control logic and contains an extensive array of charging states for each PV panel connected to the system. Given the complexity of this model, generating high-coverage test cases poses a significant challenge for traditional methods.

In particular, for the constraint solving methods, each state of the PV panels connected to the system is transformed into a formalized description. As the system accommodates a greater number of PV panels, the state space expands correspondingly. At the same time, the intricate logic of switching between various electrical energy output states has also increased the difficulty of formal solving. In addition, the execution of an embedded system should be infinitely iterative, while the constraint solver can only perform a limited loop unrolling to solve it. This limitation results in constraint solving methods often generating test cases that can only trigger shallow-level model logic. In the case of model simulation methods, the efficiency of test case generation is heavily constrained by the speed of model simulation. For this particular example model, the SimCoTest tool can only simulate the model 6 rounds per second

on average, even with Simulink's fast simulation feature enabled. This limitation makes it challenging for model simulation methods to generate high-coverage test cases within a short period of time.

To address the above challenges, we propose CFTCG, an test case generation method that combines fuzzing code generation and model oriented fuzzing loop. CFTCG first generates a fuzz driver based on the model inport information. Next, it performs branch instrumentation during the model-to-code conversion process. Then, CFTCG compiles the fuzz driver and instrumented fuzz code into a fuzz program. To generate high-quality test cases, we have designed a model oriented fuzzing loop, making it more suitable for model structures. In the model input mutation module, we incorporate information from the inports to perform field-based mutation. In the model coverage collection module, model iteration based coverage collection is utilized to identify test inputs that guide more logic to be triggered in one execution.

We implemented and evaluated CFTCG on several benchmark Simulink models. Compared to the built-in Simulink Design Verifier and the state-of-the-art academic work SimCoTest, CFTCG achieves an average improvement of 47.2% and 100.8% on Decision Coverage, 38.3% and 44.6% on Condition Coverage and 144.5% and 232.4% on Modified Condition Decision Coverage, respectively.

2 RELATED WORK

Constraint solving based test case generation. It typically employs formal techniques to obtain input cases that satisfy specific property. Simulink Design Verifier (SLDV) [5], the built-in validation toolkit of Simulink, is a popular automatic test case generation tool. It transforms the model into a formal description and then uses a formal solver to solve for test inputs that achieve the target branch coverage. He's work[6], adopt model checking approach and identify a subset of nodes that maximizes the observation of mutants. Subsequently, a small set of test cases can be generated based on this information to achieve high coverage. AutoMOTGen [10] describes the Simulink model using a formal language named SAL [7]. The coverage specifications are encoded into the formal model, and model checking tools are used to generate counterexamples for unreachable branches.

This constraint solving based test case generation method is limited by the complexity of the model's logic and states. It generally requires a formal encoding of the model and needs to be detailed down to each data bit. This means that for every additional data in the model, the difficulty of solving increases exponentially.

Model Simulation based test case generation. This approach typically utilizes dynamic simulation to obtain test feedback. SimCoTest [9] generates test cases for both continuous-time and discrete-time Simulink models. It uses meta-heuristic search to maximise the probability of specific failure modes in the output signal and to maximise the diversity of output signal shapes. Reactis [3] uses Monte Carlo methods to generate test cases through model simulation. It also uses guided simulation to evaluate output values, which helps in the selection of test cases to explore uncovered blocks. REDIRECT [12] focuses on analysing the feedback from the simulation. It uses a set of heuristics specifically designed for non-linear blocks to improve test case generation.

The limitations of model simulation based methods lie in their simulation efficiency. Although the various methods mentioned above employ unique approaches to generate high coverage test cases, they fundamentally belong to random methods. It is only when a sufficient number of test cases are generated that coverage becomes more comprehensive. However, model simulation is based on model interpretation, making it challenging to achieve efficient execution. Instead, we use the generated code and model oriented fuzzing to achieve rapid model execution, breaking free from the simulation tool's inherent speed limitation. This greatly enhances the model coverage during the test case generation process.

3 CFTCG DESIGN

Figure 2 shows an overview of CFTCG. CFTCG consists of two main parts: Fuzzing Code Generation and Model Oriented Fuzzing Loop. They all incorporate model characteristics to support efficient test case generation. In the **Fuzzing Code Generation** part, the model is initially parsed, and the inport information of the model is used to generate the fuzz driver. Subsequently, branch instrumentation is performed using the branch information obtained from the schedule conversion. This process ensures that the generated code more accurately reflects the branch structure of the model. Finally, the instrumented code will be combined with the basic code synthesis process to output the complete fuzz code. As for the **Model Oriented Fuzzing Loop** part, the fuzz driver and fuzz code generated in the previous part will be compiled into an executable program to conduct a high-speed model-oriented fuzzing loop. Within this loop, we perform targeted input mutations based on the field information of the model's input data. This model input mutation module can quickly generate new inputs suitable for target program execution. Additionally, we leverage the model iteration mechanism to collect coverage and save corpus. In this loop, the input data that causes new model coverage will be outputted as the test cases. Meanwhile, input data that achieves specific coverage metrics will be saved as interesting inputs in the corpus for the next round of mutation.

3.1 Fuzzing Code Generation.

This section mainly expands on the basic code generation workflow, i.e. Model Parser, Schedule Convert and Code Synthesis. The Model Parser module aims to provide inport information to the Fuzz Driver Generation module. The Schedule Convert module is responsible for branch instrumentation. Combined with traditional code synthesis methods, the complete fuzz code is generated.

*3.1.1 **Fuzz Driver Generation**.*

Unlike traditional fuzz drivers, the fuzz driver generated for models focuses more on the model's attributes, including its iterative execution, state dependencies between iterations, and the properties of its inports and outports. The specific process for generating the model-specific fuzz driver mainly involves the following steps:

Generating model initialization code: During model simulation, model initialization is a necessary step that must be executed before the simulation begins. Most models have their own initial states, such as the initial control states in some controller models, which can influence the subsequent logic of model execution. When generating model initialization code, we create a function that includes the initialization of model states. This function will be executed by the fuzz driver for each test input.

Generating data segmentation code: During each model iteration in simulation, the model fetches data from its inports. As for the fuzz driver, we should cyclically split the binary byte stream provided by the fuzzer into segments based on the data length obtained from the inports. The length of the data segments is calculated by summing up the data type sizes corresponding to the inports of the top-level model. Sometimes, when reaching the end of the stream, it may not contain enough data to fill all the ports. In such cases, the remaining data should be discarded, and the fuzzing of the current input data should be terminated.

Generating model execution code: We only need to call the main function corresponding to the top-level model in the fuzz driver because the functions for subsystems are also called within the main function, just like during model simulation. We further subdivide the segmented data, extracting the appropriate number of bytes according to the order of the model inports and their

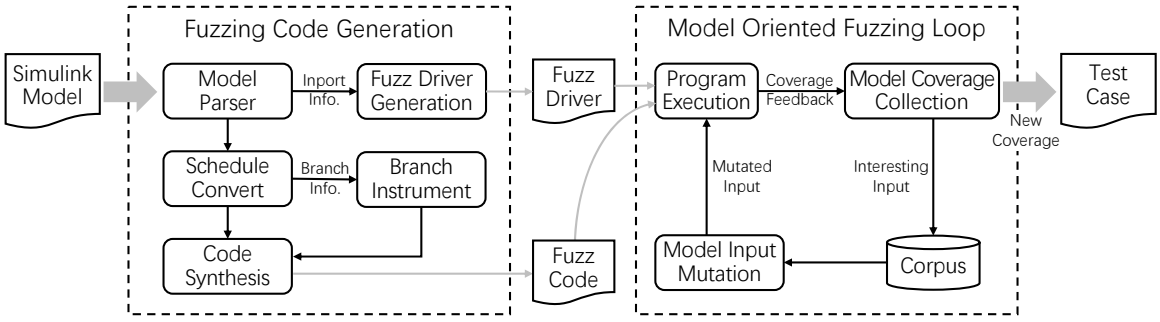


Figure 2: Overview of CFTCG. CFTCG is primarily divided into two main parts. The Fuzzing Code Generation part focuses on generating code suitable for model testing, which includes the fuzz driver and instrumented code that represents the model branch information. The Model Oriented Fuzzing Loop part concentrates on leveraging model information to achieve efficient testing, which includes input mutation and coverage collection specifically tailored to the model.

respective data types. Then, we use memory copying to assign these values to the pre-defined inport variables. After that, we pass these variables to the main function of the model for execution.

Figure 3 shows a sample fuzz driver code corresponding to the Solar PV model in Figure 1.

```
1  void FuzzTestOneInput(const uint8_t *data, size_t size){
2    SolarPV_init();          Model Initialize        Fuzz Interface
3    int dataLen = 9; Length of input data required for one iteration
4    int i = 0;
5    while(true){
6      if((i + 1) * dataLen > size){        The loop that splits one test
7        break;                             case and passes them to the
8      }                                    model for execution.
9      int8 SolarPV_Enable = {};
10     int32 SolarPV_Power = {};   Model Input
11     int32 SolarPV_PanelID = {}; Variables
12     int32 SolarPV_Ret; Model Output Variable    Model Input Assignment
13     memcpy(&SolarPV_Enable,  data + i * dataLen + 0, 1);
14     memcpy(&SolarPV_Power,   data + i * dataLen + 1, 4);
15     memcpy(&SolarPV_PanelID, data + i * dataLen + 5, 4);
16     SolarPV_step(SolarPV_Enable, SolarPV_Power,
                    SolarPV_PanelID, &SolarPV_Ret);  Model Iteration
17     i++;
18   }
19 }
```

Figure 3: Example fuzz driver for Solar PV model in Figure 1. Three variables in lines 9-11 are model inport variables. They are passed to the SolarPV_step function for model iteration.

3.1.2 *Branch Instrument.*

Model-level branch coverage provides more detailed information compared to code-level branch coverage. Any element in the model that can trigger logical decisions needs to be thoroughly accounted for in branch statistics. For this purpose, we have referred to Simulink's coverage description [14] and summarized the branch elements that require instrumentation as follows:

(a) **Boolean values of input data for boolean blocks**, such as AND, OR. The instrumentation mode involves using "$if-else$" statements to perform true/false value checks on variables corresponding to the block's inports. Within each branch statement, the coverage statistics function is inserted (Figure 4.(a)).

(b) **Different decisions for data switch/select blocks**, like Switch. The data switch block usually selects one of several data for output based on a specific condition. The instrumentation mode involves inserting coverage statistics functions into different data selection branch statements (Figure 4.(b)).

(c) **Different decisions for branch blocks**, such as If or Switch Action Subsystem. These blocks correspond to the "if" and "$switch$" statement in the code. The instrumentation mode involves adding coverage statistics functions at the beginning of all branch blocks (Figure 4.(c)).

(d) **All conditional judgments inside blocks**, such as Saturation, Matlab Function, Stateflow Chart. The instrumentation mode involves adding coverage statistics functions after all conditional statements and completing all conditional branches, including implicit "*else*" branches (Figure 4.(d)).

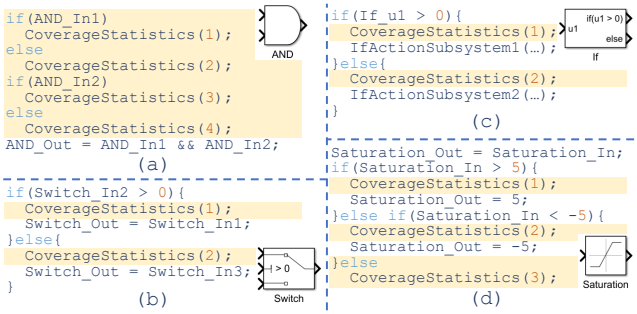


Figure 4: Examples of the four branch instrumentation modes. The code with a light yellow background indicates the generated instrumentation code.

As for the instrumentation function, the "*CoverageStatistics*()" in Figure 4, it serves as the interface for our designed fuzzer coverage collection. The detailed coverage statistics methods will be introduced later. This instrumented codes will be integrated into the basic code generation process for code synthesis. This way, we can achieve model-level coverage collection at the code level.

3.2 Model Oriented Fuzzing Loop.

To make the fuzzing loop more suitable for model testing, we have designed the model input mutation module and the model coverage collection module.

3.2.1 *Model Input Mutation.*

CFTCG performs mutations on the entire binary byte stream. Unlike the input mutation method of traditional software fuzz, we performs targeted mutation based on the model inport information, following the field-wise approach. We designed eight mutation strategies suitable for model inputs. Compared to traditional mutation strategies, CFTCG performs byte modification on tuples, which refer to collections of input data required for one model iteration. The specific mutation strategies are shown in the following table.

In Table 1, it is worth noting that the "Change Binary Integer" strategy involves modifying an integer field within a tuple, including uint8, int32, and so on. The specific modification strategies include changing the sign bit, byte swapping, bit flipping, byte modification, adding or subtracting values, and random changes.

Table 1: The strategies of model input mutation

Strategy	Description
Change Binary Integer	Modifies a binary integer value.
Change Binary Float	Modifies a binary float value.
Erase Tuples	Removes a range of tuples.
Insert Tuple	Inserts a new tuple with a random value.
Insert Repeated Tuples	Inserts a sequence of repeated tuples.
Shuffle Tuples	Shuffles the order of tuples.
Copy Tuples	Copies tuples into another position.
Tuples Cross Over	Combines tuples from two stream.

The "Change Binary Float" strategy considers targeted mutation based on the memory format of floating-point numbers. The strategies "Erase Tuples," "Insert Tuple," "Insert Repeated Tuples," "Shuffle Tuples," and "Copy Tuples" all operate on tuples, ensuring that they do not affect the validity of other data. The "Tuples Cross Over" strategy combines parts of two binary data streams to form a new binary data stream. We specifically demonstrate some mutation processes in Figure 5 to facilitate understanding.

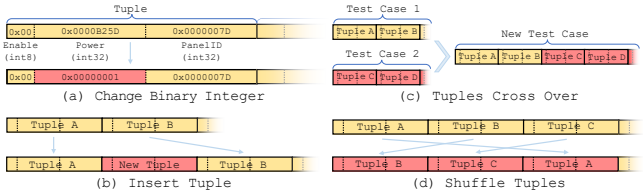


Figure 5: Example mutation strategies. The fields are corresponding to the Solar PV model in Figure 1. Each solid box represents a tuple. The dotted lines split the different fields.

3.2.2 *Model Coverage Collection*.

Based on the characteristic of model iterative execution, we have designed a new coverage metric called "Iteration Difference Coverage". This metric focuses more on the coverage differences between different iterations of the model during execution. When saving interesting inputs, we prioritize those with higher "Iteration Difference Coverage". This way, the model's execution paths can become more diversified, rather than lingering on a few main paths in each model iteration.

The detailed statistical process and method for model coverage collection are shown in Algorithm 1. For better understanding, the pseudocode is presented in the form of a loop that traverse the input tuples in the fuzz driver. Lines 6, 7, 10, and 12 represent the simplified parts of the fuzz driver. This algorithm aims to both calculate the "Iteration Difference Coverage" metric for an input data and output the inputs that trigger new model coverage as test cases. In lines 1 and 2 of Algorithm 1, we first define two arrays to separately track the overall model coverage of an input data and the individual coverage of each input tuple in every iteration. Line 3 represents the branch instrumentation function, corresponding to the "*CoverageStatistics*()" in Figure 4, used to record model branch triggers. Its invocation is implied in line 12 within the function "*Model_step*". After the execution of "*Model_step*", we calculate the total coverage and the "Iteration Difference Coverage" metric for that input data, in lines 13 to 18. In line 11, the current coverage statistics array is initialized because we need to recompute branch coverage for each round of iteration. In line 16, if new model coverage is found, we output the test case. If there is a difference between the coverage of this iteration and the previous one, we accumulate the number of differences in branch coverage to the "*metric*" variable, in lines 17 and 18. Figure 6 provides a visual sample of the statistics for the "Iteration Difference Coverage" metric.

Algorithm 1: Model Coverage Collection

Input: *data*: The binary data of one test case
size : The length of data
branchCount : Number of model branch instrumentation.
Output: *metric*: Iteration Difference Coverage metric

```
1  g_TotalCov = {0, ..., 0} //Length: branchCount
2  g_CurrCov = {0, ..., 0} //Length: branchCount
3  Function CoverageStatistcs(branchId):
4      g_CurrCov[branchId] = 1
5  End Function
6  Function FuzzTestOneInput(data, size):
7      Model_init()
8      metric = 0
9      lastCov = {0, ..., 0} //Length: branchCount
10     while tuple = getTuple(data) do
11         g_CurrCov = {0, ..., 0}
12         Model_step(tuple) //Contains calls to the CoverageStatistcs()
13         for i in {1 ... branchCount} do
14             if g_CurrCov[i] and (not g_TotalCov[i]) then
15                 g_TotalCov[i] = 1
16                 outputTestCase(data, size)
17             if g_CurrCov[i] ≠ lastCov[i] then
18                 metric = metric + 1
19         lastCov = g_CurrCov
20     return metric
21 End Function
```

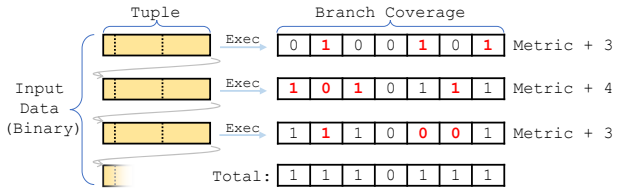


Figure 6: The statistical schematic of the "Iteration Difference Coverage" metric. It shows three iterations of the execution for an input data. On the right side, it displays the branch coverage triggered by each iteration and the total coverage after three iterations. The example's "Iteration Difference Coverage" metric is 10 (i.e., 3+4+3).

4 EVALUATION

Tool Implementation. CFTCG[1] is implemented in C++, with 44,625 lines of code. The main components include a Simulink model parser, model-to-code schedule converter, Simulink block templates, fuzz code generator, and a model-specific fuzzer. We use the Unzip and TinyXML library to load Simulink models and have developed block templates for over fifty commonly used blocks to accomplish Simulink model code generation. We have also verified the correctness of the generated code by comparing simulation results with code execution results. For the model oriented fuzzing loop, we designed it based on the LibFuzzer framework. Since LibFuzzer is an in-process fuzzer and supports Windows systems, it allows convenient test case generation for the models constructed by Simulink. For fair comparison, we implemented a tool to convert binary test case files into csv supported by Simulink for easy use with its built-in coverage statistics function.

Experiment Setup. To evaluate CFTCG, we conduct comparison experiments with the Simulink built-in validation toolkit SLDV and academic tool SimCoTest in terms of coverage results. Since other academic and commercial tools are not publicly available,

[1]The implementation and the benchmark models are uploaded on the anonymous website: https://anonymous.4open.science/r/CFTCG-138F.

we can not compare CFTCG with them. All experiments are performed on the same environment (Windows 10, Intel i7-8550U CPU, 16GB RAM) with the same duration (24 hours, but in practice, the coverage reached a stable state within an hour). We compile our fuzzing code using Clang with the O2 optimization flag. Since both SimCoTest and CFTCG include random strategies, we repeat the experiment 10 times to obtain the average coverage result for a fair comparison. All benchmark models are derived from the industry and deployed in embedded scenarios, as shown in Table 2.

Table 2: The description of benchmark models

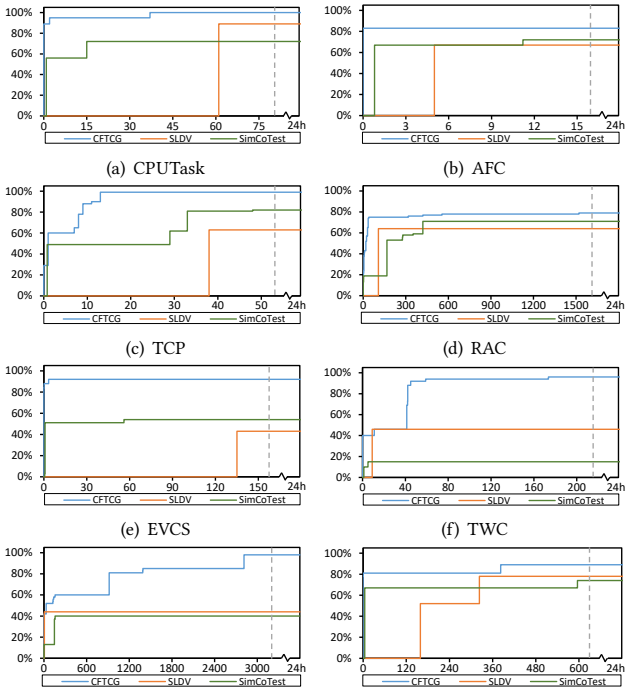
Model	Functionality	#Branch	#Block
CPUTask	AutoSAR CPU task dispatch system	107	275
AFC	Engine air-fuel control system	35	125
TCP	TCP three-way handshake protocol	146	330
RAC	Robotic arm controller	179	667
EVCS	Electric vehicle charging system	89	152
TWC	Train wheel speed controller	80	214
UTPC	Underwater thruster power control	92	214
SolarPV	Solar PV panel output control	55	131

Evaluation on Coverage Rate. We employed the widely recognized metrics of Decision Coverage, Condition Coverage, and Modified Condition Decision Coverage (MCDC) to assess the test case generation effectiveness of various tools[14]. Decision Coverage evaluates whether different branches of blocks with branch logic can be executed. Condition Coverage assesses whether conditions affecting boolean logic and branch changes are triggered. MCDC analyzes the impact of a single condition change on the entire determination. A higher coverage metric indicates a more thorough examination of the model. Table 3 shows the comparison results. Compared to SLDV and SimCoTest, CFTCG improves the Decision Coverage for 47.2% and 100.8%, Conditional Coverage for 38.3% and 44.6%, and MCDC for 144.5% and 232.4%, respectively.

Table 3: Comparison of the test coverage of different tools

Model	Tool	Decision Coverage	Condition Coverage	MCDC
CPUTask	SLDV	89%	72%	42%
	SimCoTest	72%	56%	21%
	CFTCG	**100%**	**100%**	**100%**
AFC	SLDV	67%	64%	11%
	SimCoTest	72%	68%	11%
	CFTCG	**83%**	**79%**	**22%**
TCP	SLDV	63%	64%	33%
	SimCoTest	82%	74%	17%
	CFTCG	**99%**	**96%**	**67%**
RAC	SLDV	64%	71%	12%
	SimCoTest	71%	76%	12%
	CFTCG	**79%**	**84%**	**38%**
EVCS	SLDV	80%	63%	21%
	SimCoTest	80%	63%	21%
	CFTCG	**92%**	**93%**	**83%**
TWC	SLDV	46%	68%	40%
	SimCoTest	15%	57%	20%
	CFTCG	**96%**	**98%**	**90%**
UTPC	SLDV	44%	59%	44%
	SimCoTest	40%	58%	44%
	CFTCG	**98%**	**100%**	**100%**
SolarPV	SLDV	78%	83%	57%
	SimCoTest	74%	73%	43%
	CFTCG	**89%**	**95%**	**86%**
Average Improvement	vs SLDV	↑ 47.2%	↑ 38.3%	↑ 144.5%
	vs SimCoTest	↑ 100.8%	↑ 44.6%	↑ 232.4%

We conducted further analysis on the SolarPV model which is used as an example throughout this paper. As we mentioned in the introduction section, the SolarPV model has numerous internal states and controls many solar PV panels, which leads to a huge solving space for SLDV, making it difficult to achieve higher test coverage. In the later stages of SLDV solving, its memory usage exceeded 12GB. On the other hand, we also measured the testing speed of CFTCG for this model. Compared to SimCoTest, which can only execute 6 iterations per second, CFTCG achieved a superfast speed of over 26,000 iterations per second. This significantly contributed to obtaining a more diverse set of test cases.



(a) CPUTask (b) AFC (c) TCP (d) RAC (e) EVCS (f) TWC (g) UTPC (h) SolarPV

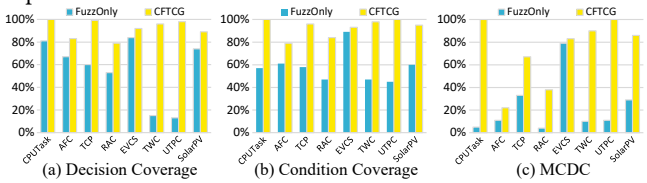
Figure 7: The folded line plot of the Decision Coverage (%, Y-axis) vs time (s, X-axis).

Efficiency of Test Case Generation. We also recorded the timestamp of each generated test case from CFTCG, SLDV, and SimCoTest. Figure 7 presents the folded line of the Decision Coverage versus time (s) for each Simulink model. From Figure 7, it is evident that in most cases, CFTCG achieves higher coverage at a faster speed and can continuously generate new test cases. Based on the experimental analysis conducted on the TWC and UTPC models, we can observe from the coverage folded line that they achieved significant coverage improvement at around 41 seconds and 917 seconds, respectively. We individually ran the two test cases that led to substantial coverage improvement, and the results indicated that they indeed triggered deep internal model states and logic.

In addition, we observed that the results on the CPUTask model reached 100% coverage, which is quite difficult for complex control models. To investigate further, we explored the specific logic of the CPUTask model and found that it has an internal task queue. Some branches are only triggered when the task queue is fullfilled. This triggering condition is very stringent for SLDV and SimCoTest. The former is limited by the complexity of solving due to state space explosion, and the latter is constrained by simulation speed. If we assume that CFTCG has the same simulation speed as SimCoTest, we estimate that it would take about 44.5 hours on average to achieve 100% coverage. However, CFTCG only took 37 seconds.

Effectiveness of Model Oriented Approach. Considering that Simulink also provides a code generator, a simple idea is to use an existing fuzzer to directly fuzz the code generated by Simulink in

order to generate test cases. We conducted the "Fuzz Only" experiment following the steps below. Firstly, we configured the Simulink code generation settings, selecting the "Reusable Function" option for code generation and the "Maximize Execution Speed" option for code optimization. "Reusable Function" option allows us to manually write the fuzz driver, similar to Figure 3. Next, we used the Clang compiler to compile the original LibFuzzer library, Simulink code, and our manually written fuzz driver together. Finally, we conducted multiple repetitions of the experiment under the same experimental environment and for the same duration as CFTCG.



(a) Decision Coverage (b) Condition Coverage (c) MCDC

Figure 8: The coverage comparison between CFTCG and the pure fuzz method ("Fuzz Only", Without model oriented).

The results are shown in Figure 8. It can be observed that CFTCG always achieves higher model coverage compared to the "Fuzz Only" method. We conducted an in-depth analysis of the experiment and found two main reasons. Firstly, the test cases generated by the "Fuzz Only" method do not cover many boolean logics. We examined the assembly code of the fuzz program compiled by the "Fuzz Only" method and found that the boolean operations did not have jump instruction and not instrumented, leading to the neglect of many Condition Coverage and MCDC. Secondly, the "Fuzz Only" method's input mutation for model testing is blind. It lacks effective generation of input data. Some models have inports containing both int8 and int32 data types, meaning that all inports of a model may not have completely consistent data types. As a result, traditional input mutation methods can cause data misalignment when deleting or inserting data in the byte stream.

5 DISCUSSION

The constraints between inports refer to situations where certain branch logics in the model requires the data of some model inports to meet specific correlated conditions to trigger certain behaviors. Fuzzing methods may not be able to detect such constraints, and we often have to rely on the random change of generating such data. Although currently, such models pose challenges to our tool and even to all existing software fuzzing methods, we are fortunate that our test targets are control models. The complexity of the models is much simpler compared to regular software, allowing us to use formal constraint solving methods to discover the relationships between ports. For instance, we can first apply constraint solving to the branches in the model to obtain the constraints between ports and then generate input data accordingly.

Validity of randomized values. Due to the development of processor and storage, model designers no longer need to be as cautious in saving computational and memory overhead when using data types. For example, an integer data might only be used within the range of 0 ~ 32768, but designers often choose to store it in an int32 variable. However, this design has a significant negative impact on fuzzing because it enlarges the space for random exploration. To address this issue, we can ask the testers to specify the value ranges for inports before test case generation. Then, during input mutation, we can add constraints based on the specified input ranges. Alternatively, if testers find it difficult to determine the value ranges for inports, we can use formal methods to determine them in advance. Based on our experimental observations, only a small number of ports in the model have explicit range constraints, such as different opcodes and some limiters for input signal values.

6 CONCLUSION

In this paper, we propose a code fuzzing based test case generation method for Simulink models. It first generates a fuzz driver that matches the model's multi-iteration execution characteristics, as well as branch instrumentation code that fulfills the model branch coverage requirements. Then, it utilizes model oriented fuzzing techniques, including input mutation based on model inport field and coverage collection based on model iterative execution.

We implemented CFTCG and conducted a detailed evaluation. Compared to SLDV and SimCoTest, CFTCG has shown significant effectiveness. The Decision Coverage from CFTCG can be improved by 47.2% and 100.8%, the Condition Coverage can be improved by 38.3% and 44.6%, and the MCDC can be improved by 144.5% and 232.4%, respectively. To further improve CFTCG, our future work involves integrating constraint solving techniques to address the related constraints between inports, and designing dynamic numerical range constraint methods to improve the mutation efficiency.

7 ACKNOWLEDGMENT

This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000), China Postdocoral Science Foundation (BX20230183, 2023M731954) and NSFC Program (No. 92167101, 62021002, 62372263).

REFERENCES

[1] Aitor Arrieta, Shuai Wang, Urtzi Markiegi, Goiuria Sagardui, and Leire Etxeberria. 2017. Search-based test case generation for cyber-physical systems. In *2017 IEEE Congress on Evolutionary Computation*. 688–697.
[2] Axel Belinfante, Lars Frantzen, and Christian Schallhart. 2005. 14 tools for test case generation. In *Model-based testing of reactive systems*. Springer, 391–438.
[3] Rance Cleaveland, Scott A Smolka, and Steven T Sims. 2006. An Anstrumentation-Based Approach to Controller Model Validation. In *Automotive Software Workshop*. Springer, 84–97.
[4] Frank Elberzhager, Alla Rosbach, and Thomas Bauer. 2013. Analysis and testing of matlab simulink models: a systematic mapping study. In *Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to Testing Automation*. 29–34.
[5] Grégoire Hamon, Bruno Dutertre, Levent Erkok, John Matthews, Daniel Sheridan, David Cok, John Rushby, Peter Bokor, Sandeep Shukla, Andras Pataricza, et al. 2008. Simulink Design Verifier-Applying Automated Formal Methods to Simulink and Stateflow. In *Third Workshop on Automated Formal Methods*.
[6] Nannan He, Philipp Rümmer, and Daniel Kroening. 2011. Test-Case Generation for Embedded Simulink via Formal Concept Analysis. In *Proceedings of the 48th Design Automation Conference*. 224–229.
[7] SRI International. 2023. *Symbolic Analysis Laboratory*. https://sal.csl.sri.com/.
[8] J Krizan, L Ertl, M Bradac, M Jasansky, and A Andreev. 2014. Automatic code generation from MATLAB/Simulink for critical applications. In *2014 IEEE 27th Canadian Conference on Electrical and Computer Engineering (CCECE)*. IEEE, 1–6.
[9] Reza Matinnejad, Shiva Nejati, Lionel C Briand, and Thomas Bruckmann. 2016. SimCoTest: A test suite generation tool for Simulink/Stateflow controllers. In *Proceedings of the 38th International Conference on Software Engineering Companion*. 585–588.
[10] Swarup Mohalik, Ambar A Gadkari, Anand Yeolekar, KC Shashidhar, and S Ramesh. 2014. Automatic test case generation from Simulink/Stateflow models using model checking. *Software Testing, Veri cation and Reliability* 24, 2 (2014), 155–180.
[11] Faruk Pasic. 2018. Model-driven development of condition monitoring software. In *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. ACM, 162–167.
[12] Manoranjan Satpathy, Anand Yeolekar, and S Ramesh. 2008. Randomized directed testing (REDIRECT) for Simulink/Stateflow models. In *Proceedings of the 8th ACM international conference on Embedded software*. 217–226.
[13] Simulink and Matlab. 2023. Simulink Documentation. (2023). https://www.mathworks.com/help/simulink/index.html.
[14] Simulink and Matlab. 2023. Simulink Model Coverage Document. (2023). https://www.mathworks.com/help/slcoverage/ug/model-coverage.html.
[15] Sergey Staroletov, Nikolay Shilov, Vladimir Zyubin, Tatiana Liakh, Andrei Rozov, Ivan Konyukhov, Innokenty Shilov, Thomas Baar, and Horst Schulte. 2019. Model-driven methods to design of reliable multiagent cyber-physical systems. In *Proc. of the Conference on Modeling and Analysis of Complex Systems and Processes*.
[16] Zehong Yu, Zhuo Su, Yixiao Yang, Jie Liang, Yu Jiang, Aiguo Cui, Wanli Chang, and Rui Wang. 2022. Mercury: Instruction Pipeline Aware Code Generation for Simulink Models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 11 (2022), 4504–4515.