AccMoS: Accelerating Model Simulation for Simulink via Code Generation

Yifan Cheng[†], Zehong Yu[‡], Zhuo Su[‡], Ting Chen[✉ †], Xiaosong Zhang[✉ †], Yu Jiang[‡]
[†] Center for Cybersecurity, University of Electronic Science and Technology of China, Chengdu, China
[‡] KLISS, BNRist, School of Software, Tsinghua University, Beijing, China

ABSTRACT

Simulink has been widely used in embedded software development, which supports simulation to validate the correctness of the constructed models. However, as the scale and complexity of models in industrial applications grow, it is time-consuming for the simulation engine of Simulink to achieve high coverage and detect potential errors, especially accumulative errors.

In this paper, we propose AccMoS, an accelerating model simulation method for Simulink models via code generation. AccMoS generates simulation functionality code for Simulink models through simulation oriented instrumentation, including runtime actor information collection, coverage collection, and calculation diagnosis. The final simulation code is constructed by composing all the instrumentation code with actor code generated from a predefined template library and integrating test cases import. After compiling and executing the code, AccMoS generates simulation results including coverage and diagnostic information. We implemented AccMoS and evaluated it on several benchmark Simulink models. Compared to Simulink's simulation engine, AccMoS shows a 215.3× improvement in simulation efficiency, significantly reduces the time required for detecting errors. AccMoS also achieved greater coverage within equivalent time.

KEYWORDS

Model-driven design, model simulation, code generation

1 INTRODUCTION

Model-driven design is widely used in embedded scenarios [7, 17, 18], which uses modeling tools like Simulink [12] to facilitate embedded software development. Although model-based development releases developers from hard-coding tasks, potential errors may occur in models and cause serious consequences, such as downcast errors, wrap on overflow, cumulative errors [19], etc. Simulation is a popular and powerful method to verify the correctness of the models and eliminate potential errors. The simulation efficiency is of vital importance, as it enables developers to discover potential errors more promptly, especially cumulative errors.

Simulink, a part of Matlab [10], is extensively utilized for modeling and simulation of embedded systems. The simulation engine of Simulink (SSE) allows for thorough verification and validation of models. It can simulate the dynamic behaviors of the target system step-by-step to identify logical errors, incorrect assumptions, and unintended behaviors within the model. Moreover, it provides runtime diagnostics to monitor the constructed model and detect potential errors. For enhanced simulation efficiency, it supports fast simulation modes, which optimizes simulation performance but simultaneously restricts the capability of runtime diagnostics and runtime information statistics.

Xiaosong Zhang and Ting Chen are the corresponding authors.

Motivation. However, SSE still falls in short to detect long-term execution errors efficiently, which often emerge after extended periods of operation. Such errors, when undetected, can lead to gradually escalating inaccuracies or system failures, potentially causing significant disruptions or damage. For example, consider the sample model shown in Figure 1. This model essentially conducts an accumulation operation on the two inputs, subsequently combining the results to produce an output. This process leads to an integer overflow error occurring at the *Sum* actor in yellow.

Using SSE, it takes 184.74s on average to detect the overflow error. However, when we manually write responsible code in C++ for this model, this error can be identified in just 0.37s averagely. This represents a speed improvement of nearly 500× compared to the SSE. The discrepancy in performance arises from Simulink's utilization of an interpreted execution method for simulation, which inherently results in slower simulation speeds. Hence, translating the model into efficient code with necessary runtime detection can substantially decrease the time required for simulation.

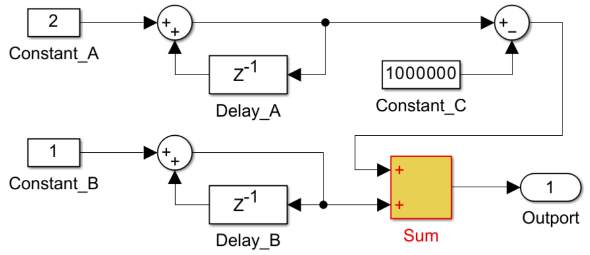

Figure 1: A sample model extracted from a large real world model, which will overflow after long time simulation.

Challenges. To accelerate model simulation in Simulink through code generation [15], we face the following two challenges. The first challenge lies in discerning the necessary data for simulation. The model of Simulink contains a vast amount of information, but not all of it is essential for simulation purposes. We specifically aim to identify the actor type and its operator for coverage analysis, while also incorporating input/output signals for the diagnostic process. Designing an effective method to extract these simulation-relevant details from the model becomes crucial.

The second challenge involves analyzing the acquired data. Comprehensive simulation functionalities, such as error diagnosis and coverage statistics, rely on the analysis of the collected data. However, the variations in actor type and its operator give rise to differences in the methods employed for error diagnosis. It is also difficult to implement coverage statistics at the model level in code. Thus, an efficient approach is required to achieve differentiation in implementing these two functionalities. Additionally, users often have specific requirements for error diagnosis, necessitating the design of a framework supporting customizing diagnostic methods.

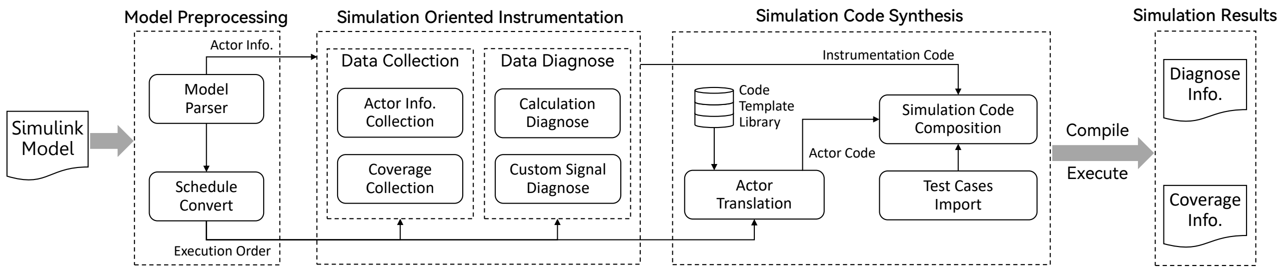

Figure 2: An overview of the AccMoS framework. AccMoS contains three key steps. The Model Preprocessing step parses the input Simulink model to collect information about the model's structure, actors, and their execution order. The Simulation Oriented Instrumentation step generates instrument code for data collection and diagnosis. The Simulation Code Synthesis step combines actor code with the previously mentioned instrumented sections to produce the final simulation code.

To address the challenges mentioned above, we introduce AccMoS, which accelerates model simulation for Simulink by translating the model into responsible code. AccMoS primarily comprises three key steps. Firstly, AccMoS parses the input Simulink model for preparation, collecting the critical information, such as the model structure, actors, and their execution order. Secondly, for each actor requiring data collection or diagnosis, AccMoS generates corresponding instrument code. After that, the actors generated in the preprocessing stage are transformed into a code referencing actor template library, and then combined with the instrumented sections to form the final simulation code. Finally, we import test cases, compile and execute all the generated code to obtain diagnostic results and coverage information.

We have implemented AccMoS and evaluated it on several benchmark Simulink models. Experimental results show that compared to SSE and its two fast simulation modes, the acceleration ratio of AccMoS reached 215.3×, 76.32×, and 19.8×, respectively. Since SSE cannot achieve error diagnosis and coverage collection in fast simulation modes, we solely compared these two functionalities of AccMoS to SSE. The coverage attained by AccMoS within equivalent time achieves substantial improvements. Also, AccMoS makes remarkable progress, reducing of the error detection time.

2 RELATED WORK

Model-driven design and Simulink. Model-driven design is a software development method that has been widely used in safety-critical embedded scenarios [4, 6, 8]. It emphasizes the use of high-level modeling and simulation to understand, visualize, and analyze the behavior of complex systems before implementation. Simulink, developed by MathWorks, is widely used in engineering, particularly for designing embedded systems and developing control algorithms. It facilitates embedded software development by supporting simulation, verification, and code generation. Among them, simulation is an effective method to verify the correctness of the constructed models and discover potential errors.

Simulation acceleration. The simulation engine (SSE) is a core part of Simulink, which enables users to execute and observe model behavior over time. It evaluates the target system step-by-step to detect logical errors, flawed assumptions, and unintended model behaviors. It is highly accurate and allows for interactive parameter tuning but can be slower for complex models. For simulation efficiency, it supports two kinds of faster modes: Accelerator mode (SSE_{ac}) and Rapid Accelerator mode (SSE_{rac}). SSE_{ac} accelerates execution by compiling the model into an intermediate MEX file, whereas SSE_{rac} entirely precompiles the model before simulation, greatly enhancing processing speed. However, these modes face limitations: frequent synchronization with Simulink and data transfer requirements may hinder speed, and their reduced error detection capabilities could compromise model accuracy and reliability. For instance, SSE_{rac} cannot detect potential errors like wrap on overflow and downcast errors, and collect coverage information.

3 DESIGN

Figure 2 shows an overview of AccMoS, involving three main steps.

The first step is model preprocessing, aiming at obtaining actors' information and their execution order from the input Simulink model. It first parses the input model to retrieve information about all the actors, and then it analyzes the execution order of all actors using a data flow labeling method [14].

The second step is simulation oriented instrumentation, focusing on generating instrumentation code for data collection and diagnostic purposes. Based on the parsed actor information, the data collection module generates code to collect runtime data of actors, involving coverage information. The data diagnose module performs diagnostic instrumentation through predefined template library. Moreover, the custom signal diagnosis sub-module allows for instrumenting user-defined diagnosis logic.

The last step is simulation code synthesis, forming the final simulation code by combining actor code with their instrumented code, as well as test cases importing code. Note that the actor code is generated based on our predefined code template library. Finally, after code compiling and executing, AccMoS obtains simulation results, including coverage information and diagnose information.

3.1 Model Preprocessing

This step takes a Simulink model file as input, and extracts information and the execution order of all the actors. The necessity for utilizing two modules arises from the characteristics of the model files. Simulink stores a model file in two main parts, involving actors and relationships. The former part contains only the fundamental information of the model, encompassing the actor's name, type, calculation operator, and the quantity of input/output signals. Note that in this part, all the actors are stored separately, with both the I/O names and data types recorded as default values with no signal connections. The relationship part stores all data flow directions, connecting I/O signals in the model.

Since then, the model parser module analyzes the actor part of the model file to gather basic information of each actor. As for the schedule convert module, we employ a directed computation graph to analyze the data flow of all signals. Then we obtain the execution order of all actors through a topological sorting technique. Meanwhile, the names and types of both input and output signals associated with each actor are extracted.

3.2 Simulation Oriented Instrumentation

In order to ensure the correctness of the model, simulation needs to detect whether errors occur in calculation actors. Additionally, to evaluate the adequacy of the testing process, it is necessary to collect coverage data in the simulation process. Since AccMoS carries out code-based simulation, code instrumentation is apparently a more suitable method to achieve these two functionalities.

Algorithm 1: Actor Code Instrumentation

```
Input: $actorInfo$: Information of all actors
       $executionOrder$ : execution order of all actors
       $collectList$ : list of actors need information collection
       $diagnoseList$ : list of actors need diagnosis
Output: $actorCode$: Instrumented actor code
1  for $actor$ in $executionOrder$ do
2      $code = genCodeFromTemp(actorInfo[actor])$
3      $diagCode = emptyString$
4      $code+ = genActorCov(actorInfo[actor])$
5      if $actorInfo[actor].isBranchActor$ then
6          $code = instConditionCov(code)$
7      if $actorInfo[actor].containBooleanLogic$ then
8          $code = instDecisionCov(code)$
9      if $actorInfo[actor].isCombinationCondition$ then
10         $code = instMCDCCov(code)$
11     if $actor$ in $collectList$ then
12         $code+ = generateCollectFunc(actorInfo[actor])$
13     if $actor$ in $diagnoseList$ then
14         $code+ = generateDiagnoseFunc(actorInfo[actor])$
15         $diagCode = genDiagnoseImpl(actorInfo[actor])$
16     $actorCode[actor].code = code$
17     $actorCode[actor].diagCode = diagCode$
18 return $actorCode$
```

The detailed process of code instrumentation is shown in Algorithm 1. The main idea of this algorithm is to instrument data collection and diagnosis code for all actors in the model. The algorithm initially traverses all actors in the order of execution, generating basic actor code for each actor based on the code template library (line 2). Afterward, the algorithm carries out relevant instrumentation operations in accordance with the characteristics specified in the actor information (lines 5-15). Note that the instrumented code here just involves the function calls at specific locations, while the actual implementation of these functions is defined elsewhere. Actor information collection functions share standardized content that can be implemented using predefined methods, as well as the coverage collection functions. However, the content of diagnostic functions varies based on the actor's type and operator, thus requiring a dynamically generated approach (line 15).

The following two sections provide detailed descriptions of the data collection module and the data diagnosis module respectively.

A. Data Collection.

Actor Info Collection. The main purpose of collecting actor information is to perform calculation diagnosis and signal monitor during simulation. In order to detect potential errors in calculation actors, it is essential to gather runtime data from each calculation actor in the model. Actor's type information is certainly required to discriminate calculation actors from the model. Additionally, diagnosis types vary depending on the type-operator combination of actors, making their operators necessary to be collected. Furthermore, the names and types of actors' input/output parameters are equally needful, since both calculation diagnosis and signal monitor require the runtime values of these parameters. Finally, to uniquely identify a specific actor within the input Simulink model and its gathered information, this module collects the actor's path as the index key, which is composed of the model file name, subsystem name, and the actor's own name, for example, *MODEL_SUBSYSTEM_ADD2*.

Figure 3 illustrates the declaration of an instrumented signal monitor function. It records the output value of the actor with three parameters, including the path of the outport, the address of its value, the corresponding data type, and the data length. All collected output values will be stored in the *outputData* data structure, which serves as a repository for result output at the conclusion of the simulation.

```
void outputCollect(string path, char* data, string type, int length) {
  outputData* OD = new outputData();
  OD->path = path;
  OD->dataType =type;
  memcpy(OD->data, data, size(type) * length);
  ...
}
```

Figure 3: An instrumented function for signal monitor.

Coverage Collection. A primary purpose of simulation is to assess the coverage of models. Coverage metrics help developers to gain deeper understanding of models' status and validate that test cases are comprehensive enough to cover different parts of models. Simulink provides four main coverage metrics [13], involving actor coverage, condition coverage, decision coverage, and modified condition/decision coverage (MC/DC). As a code-based simulation tool, AccMoS utilizes a bitmap for each metric to record runtime coverage information, which is used for coverage statistics during simulation. AccMoS attaches the instrumentation method to gather coverage information corresponding to the four coverage metrics.

(a) **Actor Coverage** indicates whether various actors in the model have been executed. We add coverage statistics code at the end of each actor, for example, `actorBitmap[actorID]=1`.
(b) **Condition Coverage** measures the executing rate beyond all the branches in the model. Conditional expressions appear in branching actors, e.g., *if*, *switch*, which determines different paths of simulation. Our method inserts coverage collection code into all executable branches.
(c) **Decision Coverage** determines the percentage of the total number of decision outcomes the code exercises during simulation. Decision points are typically associated with the actors including Boolean statements, representing different outcome values. We instrument all possible values of all the Boolean statements to collect this metric.
(d) **Modified Condition/Decision Coverage (MC/DC)** analyzes whether the conditions within a decision independently affect the decision outcome during execution. We place the instrumentation code to gather the number of conditions evaluated to all possible outcomes that impact the output of a decision. After simulation, we divide the collected values by the total number of conditions within all decisions to obtain MC/DC.

B. Data Diagnose.

Calculation Diagnose. Another primary purpose of simulation is to diagnose models for discovering various types of potential errors. Such errors are often related to computational issues that arise from the model's structure or inputs, normally appearing in calculation actors. AccMoS is capable of diagnosing all types of calculation errors supported by SSE in default, including warp on overflow, array out of bounds, division by zero, precision loss, etc. For different error types, we have developed a distinct diagnostic code and packaged them into corresponding template library. For the same error type, the instrumented diagnostic code is almost the same. Note that, the type and number of diagnoses vary depending on the actor type and its operator. For example, a "Product" actor with the "/" operator needs to diagnose division by zero errors. Conversely, when this actor uses the "*" operator, this diagnosing becomes unnecessary.

Figure 4 shows a part of the declaration of a diagnostic function, which exams a *Sum* type actor named "Minus" has the operator "-" with its runtime input/output values. Line 2 represents the diagnostic logic of detecting warp on overflow, followed by the parameter downcast diagnosis in line 4. When an error is triggered, corresponding diagnostic information will be outputted (line 3 and 5).

```
1 void diagnose_Model_Minus(i32 out, i32 in1, i32 in2) {
2   if((in1 > 0 && in2 < 0 && out < 0) || (in1 < 0 && in2 > 0 && out > 0))
3     printf("WARRING: Wrap on overflow occur on Model_Minus!\n");
4   if(sizeof(out) < sizeof(in1) || sizeof(out) < sizeof(in2))
5     printf("WARRING: Downcast may exist on Model_Minus!\n");
6   ...
7 }
```

Figure 4: A generated diagnostic function.

Custom Signal Diagnose. Sometimes users want to check whether the input/output of a certain actor meets their expectations, but a deviation from expectations does not necessarily indicate an error. In such cases, the template-based diagnosis method provided by AccMoS may not be effectively suited to handle this situation. Since then, AccMoS allows users to customize signal diagnosis, implementing their own diagnostic logic by defining callback functions. For example, detecting sudden signal changes, monitoring the output value of a specified actor, etc.

3.3 Simulation Code Synthesis

Actor Translation. Since actors of the same type share similar code, we predefine a code template library for commonly used actor types to generate the corresponding code. Notably, the same type of actors may have different detailed information, resulting in differences in the generated code. For instance, the code generated for *Math* actor varies depending on the operator it takes, e.g., exp or log. Consequently, AccMoS needs to configure such actor information to obtain the required code precisely. After that, according to the execution order, the generated code of actors is synthesized to form the mainbody code of the model.

Simulation Code Composition. In this module, the instrumentation code generated by the former step is inserted into the corresponding positions within each actor. Since the entire execution logic of the model is composed, AccMoS encapsulates it within a model system function, exemplified in the second part of Figure 5. Then AccMoS generates a main function to implement the simulation loop, where the model system function is invoked to carry

① Code of main function

```
 1 int main(int argc, char* argv[]) {
 2   TestCase_Init(); Model_Init();
 3   // Simulation Loop of model
 4   for(int step = 0; step < TOTAL_STEP; step++) {
 5     int Inport_A = takeTestCase();
 6     int Inport_B = takeTestCase();
 7     int Outport;
 8     Model_Exe(Inport_A, Inport_B, &Outport);
 9     recordResult();
10   }
11   outputResult();
12 }
```

② Code of model system function

```
1 void Model_Exe(int Inport_A, int Inport_B, int* Outport) {
2   int Minus_Out;
3   //Calculate code of Sum type actor "Model.Minus"
4   Minus_Out = Inport_A - Inport_B;
5   actorBitmap[0] = 1;
6   outputCollect("Model_Minus_out", (u8*)(&Minus_Out), "i32", 1);
7   diagnose_Model_Minus(Minus_Out, Inport_A, Inport_B);
8   ...
9 }
```

Figure 5: A sample of simulation code.

out the simulation process. An illustrative example of a main function is shown in the first part of Figure 5. In addition, in order to import test cases, the main function initializes them (line 2) before simulation and acquires the corresponding values for each input port during the simulation loop. Moreover, the code responsible for outputting simulation results (including diagnostic and coverage information) is placed at the end of the main function (line 11).

3.4 Implementation

AccMoS[1] is implemented in C++ with 36,528 lines of code. In the model preprocessing phase, the Simulink model undergoes parsing into an XML file, facilitating the generation of instrumentation code and actor code by providing actor information. To enhance diagnostic capabilities during simulation, we have meticulously developed a diagnostic code template library encompassing all error types that Simulink defaults to enable. Furthermore, specialized code template libraries have been crafted for over fifty commonly used actors, ensuring a streamlined and efficient process of code generation for Simulink models.

4 EVALUATION

To validate AccMoS, we conducted a comparative analysis against SSE with 10 Simulink benchmark models. All experiments were executed in a consistent environment (Windows 11, Intel i7-13700F CPU, 32GB RAM). Each data point represents the average of five experiment runs, ensuring the reliability and stability of the results. For comparison on error diagnosis and coverage collection, we solely compared AccMoS with SSE, as $\mathrm{SSE}_{\mathrm{ac}}$ and $\mathrm{SSE}_{\mathrm{rac}}$ cannot perform error diagnosis and coverage collection. As shown in Table 1, all benchmark models are derived from industry and deployed in embedded scenarios. The simulation code was compiled by a C/C++ Compiler (GCC 8.1.0), employing -O3 optimization flag.

Evaluation on Simulation Time. A comparative analysis was carried out to evaluate the simulation times of AccMoS, SSE and $\mathrm{SSE}_{\mathrm{rac}}$ on benchmark models. To meet the industrial requirements of long-term execution and stability tests, the simulations were conducted with a significant step size of 50 million. The results shown in Table 2 illustrate that AccMoS achieves significant performance

[1]The implementation of AccMoS and benchmark models are available at the anonymous website: https://anonymous.4open.science/r/AccMoS-C0CC.

improvement. Compared to SSE, SSE_{ac} and SSE_{rac}, AccMoS displayed an average efficiency improvement of 215.3×, 76.32× and 19.8×, respectively.

Table 1: The description of benchmark models

Model	Functionality	#Actor	#SubSystem
CPUT	AutoSAR CPU task dispatch system	275	27
CSEV	Charging system of electric vehicle	152	17
FMTM	Factory Multi-point Temperature Monitor	276	42
LANS	LAN Switch controller	570	39
LEDLC	LED light controller	170	31
RAC	Robotic arm controller	667	57
SPV	Solar PV panel output control	131	16
TCP	TCP three-way handshake protocol	330	42
TWC	Train wheel speed controller	214	13
UTPC	Underwater thruster power control	214	21

We observe that the acceleration ratios of four models, namely LANS, LEDC, SVP, and TCP, are significantly higher than other models, compared with SSE. By conducting an in-depth analysis of these model structures, we found that they contain more computational actors than other models. The interpretative execution method of SSE requires a substantial amount of time to process computational logic. However, for code-based simulation methods, including AccMoS and SSE_{rac}, the code for computational operations benefits from compiler optimizations and processor features like pipelining and superscalar architectures, enabling faster simulation. On the other hand, code generated from control logic actors, which includes conditional statements, is less amenable to such optimizations by compilers or processors. Consequently, higher acceleration ratios are achieved in these models.

Table 2: Comparison of simulation time

Model	AccMoS	SSE	SSE_{ac}	SSE_{rac}	Improvement		
					SSE	SSE_{ac}	SSE_{rac}
CPUT	4.21s	167.67s	69.55s	37.41s	39.8×	16.5×	8.9×
CSEV	0.77s	75.06s	43.97s	35.58s	97.5×	57.1×	46.2×
FMTM	2.42s	70.61s	58.31s	32.80s	29.2×	24.1×	13.6×
LANS	3.61s	1603.21s	536.81s	99.96s	444.1×	148.7×	27.7×
LEDLC	4.31s	1688.20s	512.75s	48.66s	391.7×	119.0×	11.3×
RAC	3.45s	108.99s	70.77s	48.35s	31.6×	20.5×	14.0×
SPV	1.67s	934.88s	375.66s	34.60s	559.8×	224.9×	20.7×
TCP	2.09s	768.05s	158.26s	46.15s	367.5×	75.7×	22.1×
TWC	2.05s	182.27s	76.22s	41.34s	88.9×	37.2×	20.2×
UTPC	10.88s	1120.77s	430.06s	140.38s	103.0×	39.5×	12.9×

While SSE_{ac} employs a strategy of compiling models into MEX files to reduce the interpretive execution overhead, thus boosting simulation efficiency, it still relies on interpretive execution for simulations. Consequently, AccMoS significantly outperforms SSE_{ac} in terms of simulation efficiency. As for SSE_{rac}, it precompiles the target model and employs code-based simulation method to accelerate the simulation efficiency. However, its performance is still constrained by the need for frequent synchronization and data transfer with Simulink, which poses a limitation to achieving optimal simulation efficiency.

Effectiveness of Coverage Collection. We conducted a comparative analysis between AccMoS and SSE, with equivalent test cases generated through a random approach. The evaluation specifically centered on comparing the coverage achieved by both methodologies within a consistent simulation time frame. Coverage metrics, including actor, condition, decision, and MC/DC, were systematically recorded at simulation intervals of 5s, 15s, and 60s. Detailed results are presented in Table 3.

Coverage metrics are essential in model-driven development, helping developers gain a deeper understanding of the model's execution status and validating the comprehensiveness of tests. Attaining high coverage more quickly further aids developers in efficiently analyzing the model. Our experiments indicate that within just 5 seconds, all 4 coverage metrics achieved by AccMoS surpass 60 seconds of SSE simulation, for all models apart from the TCP model. As for TCP, after a very brief 15-second simulation, its coverage comprehensively surpassed the results obtained through simulation on SSE. AccMoS demonstrates significant efficiency improvement in coverage collection.

Table 3: Coverage of AccMoS and SSE

Model	Time (s)	Actor		Condition		Decision		MC/DC	
		AccMoS	SSE	AccMoS	SSE	AccMoS	SSE	AccMoS	SSE
CPUT	5	32%	9%	50%	13%	53%	14%	33%	7%
	15	43%	20%	76%	28%	78%	30%	64%	15%
	60	52%	20%	52%	28%	93%	30%	93%	15%
CSEV	5	46%	46%	70%	63%	69%	64%	45%	33%
	15	46%	46%	73%	63%	71%	64%	50%	33%
	60	46%	46%	73%	66%	71%	67%	50%	38%
FMTM	5	37%	2%	49%	3%	48%	2%	25%	0%
	15	45%	10%	60%	10%	57%	10%	31%	2%
	60	45%	10%	62%	10%	59%	10%	36%	2%
LANS	5	45%	18%	62%	27%	60%	27%	37%	18%
	15	45%	45%	62%	60%	60%	58%	37%	34%
	60	45%	45%	65%	60%	62%	58%	42%	34%
LEDLC	5	51%	31%	83%	40%	82%	42%	59%	27%
	15	51%	31%	84%	43%	84%	45%	62%	35%
	60	51%	31%	85%	43%	85%	46%	64%	39%
RAC	5	43%	2%	59%	3%	55%	2%	32%	0%
	15	43%	10%	60%	11%	57%	10%	35%	2%
	60	44%	25%	62%	30%	59%	29%	38%	12%
SPV	5	49%	44%	84%	63%	83%	63%	68%	40%
	15	49%	44%	84%	73%	83%	72%	68%	56%
	60	49%	44%	84%	73%	83%	72%	68%	56%
TCP	5	40%	23%	65%	25%	65%	24%	58%	10%
	15	40%	24%	65%	26%	65%	25%	58%	13%
	60	40%	37%	65%	58%	65%	58%	58%	50%
TWC	5	38%	21%	59%	36%	55%	32%	41%	16%
	15	38%	21%	59%	36%	55%	32%	41%	18%
	60	53%	38%	99%	56%	99%	52%	98%	36%
UTPC	5	38%	20%	58%	22%	57%	19%	37%	1%
	15	38%	38%	58%	55%	57%	53%	37%	28%
	60	38%	38%	61%	57%	59%	55%	43%	34%

Error Diagnosis Case Study. To demonstrate the AccMoS's capability of error detection, we manually inject errors into the CSEV model. CSEV represents an charging system of electric vehicles. It supports various modes of charging and offers different charging powers. This system has a data-store memory actor *quantity*, which represents global variable in code, to record the quantity of charged electricity, with the data type being int.

Specifically, two specific errors are intentionally injected in the CSEV model. The first error is a wrap on overflow in the *quantity* variable. This error arises during ongoing simulations, which represents the electric vehicle's continuous charging process. As a result, the value of *quantity* progressively increases, eventually exceeding the maximum limit of an integer, thus leading to an overflow. To detect this error, AccMoS employs the diagnosis code to monitor the add actor before *quantity*, using the following condition: `if(input1 > 0 && input2 > 0 && output < 0)`.

The second error involves a wrap on overfllow in the calculation of charging power. CSEV, depending on the charging mode, offers varied charging powers. It first retrieves the rated voltage and current based on the selected charging mode, and then employs a product actor to determine the charging power. However, a discrepancy arises as the output data type of this product actor is short int, differing from the int data type of voltage and current, resulting in a wrap on overfllow error. To identify this error, AccMoS employs the `sizeof()` function to determine the data sizes of both the inputs and outputs in the product calculation. A wrap on overfllow error is indicated if these sizes do not align.

The first wrap on overflow is detected by AccMoS in just 0.74s of simulation, reducing over 99% of detection time, compared to 450.14s taken by SSE. This significant improvement shows the effectiveness of AccMoS. As for the second error, it manifests at the beginning of the simulation. Consequently, the difference in detection times is minimal, ranging between 0.18s and 1.2s.

5 DISCUSSION

Threats to validity. At present, AccMoS mainly supports code-based simulation for discrete models, but a key limitation of AccMoS is its current lack of capability in supporting continuous models [5, 11]. In contrast to discrete models, which experience changes at specific intervals, continuous models represent variations occurring continuously over any given time frame. Expanding AccMoS to encompass both discrete and continuous models would significantly enhance its versatility and utility. To support code-based simulation of continuous models, AccMoS could integrate numerical solvers, such as Adams solver [2], to effectively resolve differential equations inherent in these continuous models.

Extensibility of AccMoS. AccMoS currently focuses on accelerating the simulation process of Simulink models. Generally, there are other well-known model-driven tools, also widely used in embedded software development, such as Ptolemy-II, SCADE, and Tsmart [1, 3, 9]. To support code-based simulation for these tools, AccMoS must be capable of parsing their unique model representations and then generating the corresponding simulation code. One possible way to address this problem is to build a well-structured intermediate representation (IR) that ensures compatibility with various model-driven design tools. Additionally, to further enhance simulation efficiency, AccMoS could explore leveraging optimization techniques used by other code generators [16, 20].

6 CONCLUSION

In this paper, we have presented AccMoS, a novel approach to accelerate model simulation for Simulink through automated code generation. AccMoS works by first preprocessing the given Simulink model, and then generating instrumentation code for simulation functionality. The final code is then synthesized by integrating this instrumentation code with the actor code generated from templates, as well as test cases importing code. Through code-based simulation, AccMoS rapidly produces results containing coverage and diagnostic information.

We implemented AccMoS and evaluated it on several benchmark Simulink models. The results demonstrate that compared to SSE, AccMoS achieves a substantial simulation accelerating ratio up to 215.3×, significantly reducing the time required for error diagnosing, as well as remarkable coverage collection ability. In the future, we plan to extend AccMoS's capabilities for supporting continuous models and other modeling environments.

7 ACKNOWLEDGMENT

This research is sponsored in part by the China Postdocoral Science Foundation (BX20230183, 2023M731954), the NSFC (U2336204, 62332004, 62372263, 92167101, 62021002), the National Key Research and Development Project (2022YFB3104000), and the Sichuan Natural Science Foundation Project (24ZNSFSC0038).

REFERENCES

[1] Gérard Berry. 2007. SCADE: Synchronous design and validation of embedded control software. In *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*. Springer, 19–33.

[2] Tomas Brezina, Zdenek Hadas, and Jan Vetiska. 2011. Using of Co-simulation ADAMS-SIMULINK for development of mechatronic systems. In *14th International Conference Mechatronika*. IEEE, 59–64.

[3] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. 2002. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. In *Readings in Hardware/Software Co-Design*, Giovanni De Micheli, Rolf Ernst, and Wayne Wolf (Eds.). Morgan Kaufmann, San Francisco, 527–543.

[4] Peter H Feiler. 2010. Model-based validation of safety-critical embedded systems. In *2010 IEEE Aerospace Conference*. IEEE, 1–10.

[5] W Michael Hanemann. 1984. Discrete/continuous models of consumer demand. *Econometrica: Journal of the Econometric Society* (1984), 541–561.

[6] Hans Hansson, Mikael Åkerholm, Ivica Crnkovic, and Martin Torngren. 2004. SaveCCM-a component model for safety-critical real-time systems. In *Proceedings. 30th Euromicro Conference, 2004.* IEEE, 627–635.

[7] David Hästbacka, Timo Vepsäläinen, and Seppo Kuikka. 2011. Model-driven development of industrial process control applications. *Journal of Systems and Software* 84, 7 (2011), 1100–1113.

[8] Yu Jiang, Han Liu, Houbing Song, Hui Kong, Rui Wang, Yong Guan, and Lui Sha. 2018. Safety-assured model-driven design of the multifunction vehicle bus controller. *IEEE Transactions on Intelligent Transportation Systems* 19, 10 (2018), 3320–3333.

[9] Yu Jiang, Hehua Zhang, Huafeng Zhang, Xinyan Zhao, Han Liu, Chengnian Sun, Xiaoyu Song, Ming Gu, and Jiaguang Sun. 2014. Tsmart-galsblock: A toolkit for modeling, validation, and synthesis of multi-clocked embedded systems. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 711–714.

[10] Mathworks. 2023. MATLAB. (2023). https://www.mathworks.com/help/matlab/index.html.

[11] Reza Matinnejad, Shiva Nejati, Lionel C Briand, and Thomas Bruckmann. 2016. Automated test suite generation for time-continuous simulink models. In *proceedings of the 38th International Conference on Software Engineering*. 595–606.

[12] Simulink and Matlab. 2023. Simulink Documentation. (2023). https://www.mathworks.com/help/simulink/index.html.

[13] Simulink and Matlab. 2023. Simulink Model Coverage Document. (2023). https://www.mathworks.com/help/slcoverage/ug/model-coverage.html.

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

[15] Zhuo Su, Dongyan Wang, Yixiao Yang, Zehong Yu, Wanli Chang, Wen Li, Aiguo Cui, Yu Jiang, and Jiaguang Sun. 2021. MDD: A unified model-driven design framework for embedded control software. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 10 (2021), 3252–3265.

[16] Zhuo Su, Zehong Yu, Dongyan Wang, Yixiao Yang, Yu Jiang, Rui Wang, Wanli Chang, and Jiaguang Sun. 2022. HCG: Optimizing embedded code generation of Simulink with SIMD instruction synthesis. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 1033–1038.

[17] Kleanthis Thramboulidis, D Perdikis, and S Kantas. 2007. Model driven development of distributed control applications. *The International Journal of Advanced Manufacturing Technology* 33 (2007), 233–242.

[18] Gabriele Trombetti, Aniruddha Gokhale, Douglas C Schmidt, Jesse Greenwald, John Hatcliff, Georg Jung, and Gurdip Singh. 2005. An integrated model-driven development environment for composing and validating distributed real-time and embedded systems. *Model-Driven Software Development* (2005), 329–361.

[19] Eric-Jan Wagenmakers, Peter Grünwald, and Mark Steyvers. 2006. Accumulative prediction error and the selection of time series models. *Journal of Mathematical Psychology* 50, 2 (2006), 149–166.

[20] 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.