AccSiM: State-Aware Simulation Acceleration for Simulink Models

Yifan Cheng, Zehong Yu, Zhuo Su, Ting Chen, *Member, IEEE*, Xiaosong Zhang, and Yu Jiang

***Abstract*—Simulink has been widely used in embedded software development, which supports simulation to validate the correctness of 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 article, we propose AccSiM, an accelerating model simulation method for Simulink models via code generation. AccSiM generates simulation functionality code for Simulink models through simulation oriented instrumentation, including runtime data collection, data diagnosis, and state-aware acceleration. 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, AccSiM generates simulation results including coverage and diagnostic information. We implemented AccSiM and evaluated it on several benchmark Simulink models. Compared to Simulink's simulation engine, AccSiM shows a 215.3× improvement in simulation efficiency, significantly reduces the time required for detecting errors. Furthermore, through the state-aware acceleration method, AccSiM yielded an additional 2.8× speedup. AccSiM also achieved greater coverage within equivalent time.**

***Index Terms*—Code generation, model simulation, model-driven design.**

I. Introduction

MODEL-DRIVEN development (MDD) has been widely used in industrial applications [2], [3], [4], [5], [6], which uses modeling tools like Simulink [7] and open-source Ptolemy-II [8], [9], [10] to facilitate software development. It is prevalent not only in traditional embedded software design [11], [12], [13], but also increasingly adopted in emerging areas like the Internet of Things, artificial intelligence, and cloud computing for system design [14], [15], [16], [17], [18]. MDD involves the processes of modeling, simulation, testing, and code generation to transform high-level abstract data flow models into executable code, thereby enhancing development efficiency and ensuring system consistency and reliability.

Although MDD offers these significant advantages, it does not eliminate errors within the models. First, errors introduced by developers during the modeling process may lead to inconsistencies between the model's functionality and the requirements. Second, the complexity of large industrial models may make it challenging for developers to fully grasp all the intricate details of the system, thereby potentially overlooking errors or boundary conditions, such as overflow errors in model computation actors or accumulation errors resulting from prolonged execution [19], [20]. Since the industrial applications of MDD are often safety-critical, including those in industrial, vehicle, and satellite control systems, even occasional errors in these scenarios could lead to severe accidents and substantial losses [21], [22], [23]. Therefore, eliminating potential errors in models is crucial.

Model simulation stands as one of the primary approaches to identify and resolve errors within models. 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, computational errors, and incorrect assumptions within the model. Moreover, it provides runtime diagnostics to monitor the constructed model and detect potential errors. For enhancing simulation efficiency, SSE supports fast simulation modes, which optimizes simulation performance but restricts the capability of runtime diagnostics and information collection.

Motivation: However, SSE still falls in short to detect long-term execution errors efficiently due to its interpreted execution method, 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 Fig. 1. This model essentially begins by evaluating the input data; if the data does not meet the specified condition, the output maintains 0. If the conditions are met, the model performs an accumulation operation on the two inputs, subsequently combining the results to produce output. This process leads to an integer overflow error occurring at the *Sum* actor in yellow.

Received 31 August 2024; revised 9 January 2025; accepted 19 February 2025. Date of publication 28 February 2025; date of current version 22 August 2025. This work was supported in part by the China Postdoctoral Science Foundation under Grant BX20230183 and Grant 2023M731954; in part by the National Natural Science Foundation of China under Grant U2336204, Grant 62332004, Grant 62372263, Grant 92167101, Grant 62021002, Grant 62372086, and Grant 62472075; in part by the National Key Research and Development Project under Grant 2022YFB3104000 and Grant 2023YFB3105900; in part by the Natural Science Foundation of Sichuan under Grant 2024NSFSC0004; and in part by the Sichuan Provincial Natural Science Foundation for Distinguished Young Scholars under Grant 2023NSFSC1963. This paper is an extended version of a conference paper [1] [DOI: 10.1145/3649329.3656218]. This article was recommended by Associate Editor R. Doemer. *(Corresponding authors: Ting Chen; Yu Jiang; Zhuo Su.)*

Yifan Cheng, Ting Chen, and Xiaosong Zhang are with the Center for Cybersecurity, University of Electronic Science and Technology of China, Chengdu 610054, China (e-mail: brokendragon@uestc.edu.cn).

Zehong Yu, Zhuo Su, and Yu Jiang are with the KLISS, BNRist, School of Software, Tsinghua University, Beijing 100084, China (e-mail: suzcpp@gmail.com; jiangyu198964@126.com).

Digital Object Identifier 10.1109/TCAD.2025.3546879

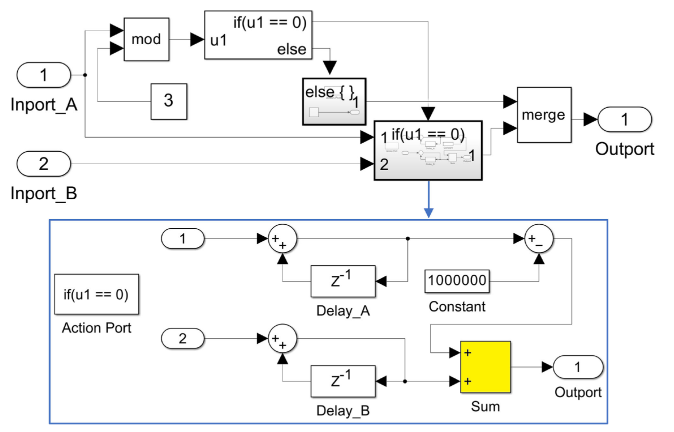


Fig. 1. Sample model extracted from a large real world model, which will overflow at the highlighted Sum actor after long time simulation.

We used a randomly generated sequence of positive integers as input to simulate the model. Using SSE, it takes 217.58 s to detect the overflow error. However, when we manually write the responsible code in C++ for this model, this error can be identified in just 0.43 s. This represents a speed improvement of over 500× compared to SSE. The discrepancy in performance arises from Simulink's utilization of an interpreted execution method for simulation, which includes data type resolution, dynamic resolution of actor parameters, and dynamic scheduling of actors [7]. Hence, translating the model into efficient code with necessary runtime detection can substantially decrease the time required for simulation. Additionally, from the model structure in Fig. 1, we can observe that the accumulation operation only occurs when `Inport_A % 3 = 0`, while other two-thirds (the *else* actor) computation being duplicate. By skipping these duplicate simulation loops, we could achieve an additional threefold increase in simulation speed. To accelerate model simulation in Simulink through code generation [24], we face the following three challenges.

Challenge 1 (Identifying Essential Simulation Data): The initial hurdle is to pinpoint the indispensable data required for simulation within the Simulink model. Although the model encompasses a rich reservoir of information, not all of it is pertinent for simulation purposes. Our primary focus revolves around discerning the actor type and its corresponding operator for comprehensive coverage analysis. Additionally, it is imperative to integrate input and output signals into the diagnostic process seamlessly. Thus, formulating a robust methodology for extracting these simulation-relevant details from the model emerges as a pivotal task.

Challenge 2 (Analyzing Acquired Data): The subsequent challenge entails delving into the acquired data. The effectiveness of comprehensive simulation functionalities, such as error diagnosis and coverage statistics, hinges on the meticulous analysis of the amassed data. However, the diverse array of actor types and their corresponding operators precipitates discrepancies in the methodologies deployed for error diagnosis. Moreover, grappling with the intricacies of implementing coverage statistics at model level poses a formidable challenge. Consequently, a streamlined approach is imperative to delineate the implementation of these two functionalities distinctly. Furthermore, users often harbor specific requisites concerning error diagnosis, necessitating the development of a framework conducive to customizing diagnostic methodologies.

Challenge3 (Skipping Duplicate Iteration): In practical industrial scenarios, where system states and inputs for models are often restricted, employing a state-aware approach can expedite simulations by avoiding redundant execution of identical simulation steps. The key challenge lies in discerning which executions can be safely bypassed. This necessitates real-time monitoring and analysis of the model's state, coupled with the design of efficient decision-making mechanisms to ascertain the necessity of executing the current simulation loop. However, achieving this objective without compromising accuracy and reliability poses a nontrivial task. It requires a thorough consideration of multiple factors and skillful management of the complexities that arise from the dynamic nature of the model and variations in input parameters. Furthermore, the introduction of this method will inevitably incur significant overhead, necessitating the use of sophisticated algorithms to minimize it as much as possible.

To address the challenges mentioned above, we introduce AccSim, which accelerates model simulation for Simulink by translating the model into responsible code. AccSim primarily comprises three key steps. First, AccSim parses the input Simulink model for preparation, collecting the critical information, such as the model structure, actors, and their execution order. Second, for each actor requiring data collection or diagnosis, AccSim generates corresponding instrument code. Additionally, AccSim offers optional state-aware acceleration approach, which is an optimization strategy based on model states, aimed at enhancing simulation efficiency. By precisely tracking and identifying model states, this method can skip redundant simulation loops, thereby reducing unnecessary computational overhead. 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 compile and execute all the generated code with imported test cases to obtain diagnostic results and coverage information.

We have implemented AccSim 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 AccSim reached 215.3×, 76.32×, and 19.8×, respectively. Besides, acceleration through the state-aware method yielded an additional 2.8× speedup. Since SSE cannot achieve error diagnosis and coverage collection in fast simulation modes, we solely compared these two functionalities of AccSim to SSE. The coverage attained by AccSim within equivalent time achieves substantial improvements. Furthermore, AccSim makes remarkable progress reducing the error detection time.

To sum up, our main contributions are as follows.

1) We propose AccSim, an accelerating model simulation method for Simulink models via code generation.
2) We introduce a state-aware simulation acceleration method to boost simulation performance by skipping duplicate simulation loops.

3) We implement AccSiM and evaluate its effectiveness and performance on benchmarks. The results demonstrate that AccSiM outperforms the state-of-the-art simulation tool SSE by 215.3×, with better error diagnose capacity.

II. Background

A. Model-Driven Design

MDD [25] is a software development approach that employs data flow models as its primary development semantics to enhance software development efficiency, reduce development costs, and improve software quality. Its core concept lies in utilizing formal models to describe the structure and behavior of systems, enabling developers to understand and analyze systems more intuitively, thereby accelerating development speed, reducing development costs, and enhancing software quality. The central idea of MDD is to place models at the center of software development, achieving comprehensive automation from requirements analysis to system implementation. By establishing clear mappings between models and code, corresponding code auto-generation can be realized, reducing manual coding effort, lowering error rates, and improving production efficiency. At present, the main development environments used for MDD include Simulink in MATLAB and the open-source Ptolemy-II framework.

B. Embedded System Execution and Simulation

In embedded systems, the main loop typically serves as the core of the control software, responsible for periodically handling task scheduling, sensor readings, data processing, and control command execution. The execution time of the main loop is influenced by factors such as the hardware platform, software algorithms, and external environmental conditions. Consequently, the corresponding real-world execution time needs to be determined based on the specific application scenario of the system, considering factors like external interrupts and resource contention.

To evaluate the functionality and performance of embedded systems, simulation methods are commonly employed to model their behavior [26], [27]. Simulation loops simulate the system's time progression using specialized tools, with each simulation step representing a discrete time interval of the system's state. The granularity of the simulation determines the time precision of each simulation update. The corresponding real-world time for the simulation is calculated based on the specific characteristics of the system and its environment. For instance, factors such as hardware performance, resource availability, and the frequency of real-time parameter acquisition influence this calculation, highlighting the importance of tailoring simulations to the application context.

C. Simulink Simulation Engine

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, SSE 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} can neither detect potential errors like overflow and downcast, nor collect coverage information.

D. Coverage Metrics

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 [28], involving actor coverage, condition coverage, decision coverage, and modified condition/decision coverage (MC/DC).

Actor coverage indicates whether various actors in the model have been executed.

Condition coverage measures the executing rate beyond all the branches in the model. Conditional expressions appear in branching actors, e.g., *if*, *switch*, which determine different paths of simulation.

Decision coverage determines the percentage of the total number of decision outcomes the code executes during simulation. Decision points are typically associated with the actors including Boolean statements, representing different possible outcome values.

MC/DC analyzes whether the conditions within a decision independently affect the decision outcome during execution.

III. Design

Fig. 2 shows an overview of AccSiM, 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 [29]. 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. The state-aware acceleration module is responsible for generating instrumentation code, which collects system state, input, and output information during runtime, and based on this information, decides whether to execute the simulation process for the current input or directly update the model state and output information, and skip the current input. Moreover, the custom signal diagnosis submodule allows for instrumenting

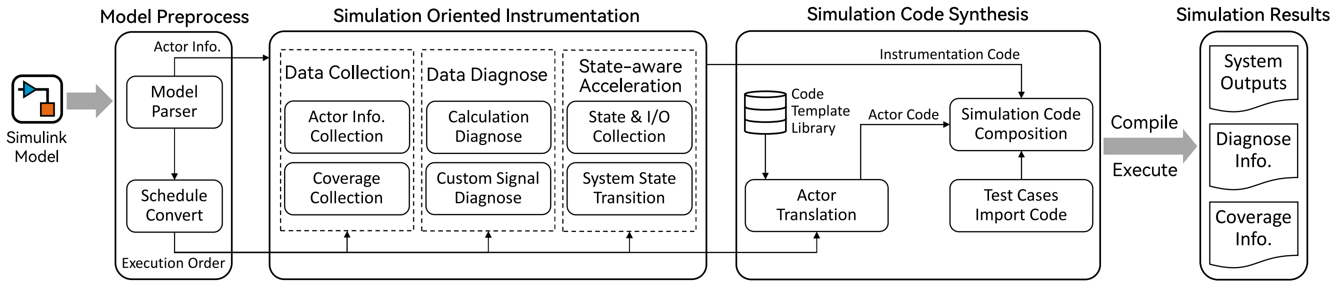


Fig. 2. Overview of the AccSim framework. The Model Preprocess step parses the input Simulink model to collect information about the model's structure, actors and their execution order. The Simulation Oriented Instrumentation step generates instrument code for data collection, data diagnosis and state-aware acceleration. The Simulation Code Synthesis step combines actor code with the instrumented sections to produce the final simulation code.

TABLE I
Simulation Features and Their Relevant Model Information

Model Information	Simulation Features			
	Signal Monitor	Data Diagnosis	Coverage Statistics	State-Aware Acceleration
Actor Name	✓	✓	✓	-
Actor Type	-	✓	✓	-
Operator	-	✓	-	-
I/O Signal Name	✓	✓	-	-
I/O Data Type	✓	✓	-	-
Global Variable	-	-	-	✓
Model Hyperparameter	-	-	-	✓
Execution Order	-	✓	✓	-

user-defined diagnosis logic dynamically. The last step is simulation code synthesis, forming the final simulation code by combining actor code with their instrumented code, as well as the test cases importing code. Note that the actor code is automatically generated based on our predefined code template library.

A. Model Preprocessing

This step takes a Simulink model file as input, and extracts information and the execution order of all the actors to support simulation oriented instrumentation and model code generation in subsequent steps. 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.

To effectively address the challenge of identifying simulation-relevant details, it is essential to establish a clear definition of these details. Each simulation feature corresponds to specific model-related information, as summarized in Table I. A comprehensive explanation of how this information facilitates various simulation features will be provided in the subsequent sections.

Given that Simulink model files are stored in XML format, the model parser module leverages the TinyXML library to thoroughly interpret the actor part. This process extracts fundamental information for each actor, including the actor's name, type, associated operator, and default I/O signal specifications. The schedule convert module further processes the relationship part of the model by representing the data flow as a directed computation graph. In this graph, nodes represent actors, and edges denote signal connections. This representation enables data flow analysis to establish precise interconnections and employs topological sorting to determine the execution order of all actors systematically.

B. 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 AccSim carries out code-based simulation, code instrumentation is apparently a more suitable method to achieve these two functionalities.

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, typically in other files or libraries. Actor information collection functions share standardized content that can be implemented using predefined methods, as well as the coverage collection functions. However, the specific content of diagnostic functions varies significantly based on the actor's type and operator, thus requiring a dynamically generated approach (line 15).

1) 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

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
for $actor$ in $executionOrder$ do
    $code = genCodeFromTemp(actorInfo[actor])$
    $diagCode = emptyString$
    $code += genActorCov(actorInfo[actor])$
    if $actorInfo[actor].isBranchActor$ then
        $code = instConditionCov(code)$
    if $actorInfo[actor].containBooleanLogic$ then
        $code = instDecisionCov(code)$
    if $actorInfo[actor].isCombinationCondition$ then
        $code = instMCDCCov(code)$
    if $actor$ in $collectList$ then
        $code += generateCollectFunc(actorInfo[actor])$
    if $actor$ in $diagnoseList$ then
        $code += generateDiagnoseFunc(actorInfo[actor])$
        $diagCode = genDiagnoseImpl(actorInfo[actor])$
    $actorCode[actor].code = code$
    $actorCode[actor].diagCode = diagCode$
return $actorCode$
```

```c
void outputCollect(string path, char* dataAddr, string dataType) {
  outputData* OD = new outputData();
  OD->path = path;
  OD->dataType = dataType;
  memcpy(OD->data, dataAddr, sizeof(dataType));
  ...
}
```

Fig. 3. Instrumented function for signal monitor.

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*.

Fig. 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* object, which serves as a repository for result output at the conclusion of the simulation.

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 [28], involving actor coverage, condition coverage, decision coverage, and MC/DC. As a code-based simulation tool, AccSiM utilizes a bitmap for each metric to record runtime coverage information, which is used for coverage statistics during simulation.

```c
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 }
```

Fig. 4. Generated diagnostic function for a Minus actor.

AccSiM attaches the instrumentation method to gather coverage information corresponding to the four coverage metrics: 1) For actor coverage, we add coverage statistics collection code at the end of each actor, for example, `actorBitmap[actorID]=1`; 2) For condition coverage, our method inserts coverage collection code into all executable branches, e.g., *if*, *switch*; 3) For decision coverage, we instrument all possible values of all the Boolean statements to collect this metric; and 4) For MC/DC, 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.

2) 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. AccSiM 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.

Fig. 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).

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 AccSiM may not be effectively suited

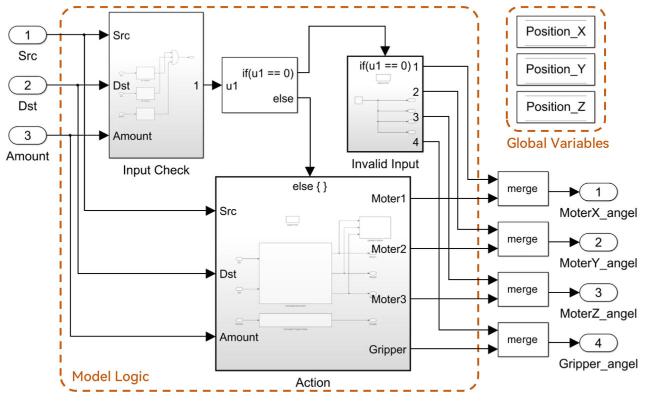


Fig. 5. Material transfer robotic arm control model extracted from a real industrial system.

to handle this situation. Since then, AccSim 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) State-Aware Acceleration: In the process of simulating real industrial models using AccSim, we noticed that many steps in the model simulation process can be skipped without affecting the simulation results. For example, Fig. 5 presents a material transfer robotic arm control model extracted from a real industrial system. It contains three inputs: the source port, the destination port, and the amount of materials. The current 3-D spatial position of the robotic arm is recorded in the global variables Position_X, Position_Y, and Position_Z. The output of the model consists of the rotational angles of the motors controlling three directions of the robotic arm, as well as the opening angle of its gripper. Whenever an input arrives, the model first calculates the displacements of the robotic arm in the three directions based on the current position recorded in the global variables and the material source port information from the input. After grabbing the material from the input port, the model calculates the displacements of the robotic arm in the three directions based on the destination port information, controlling the robotic arm to move the material to the output port. Finally, based on the amount of input materials, it determines the opening angle of its gripper. After completion, the new position of the robotic arm is recorded in the global variables Position_X, Position_Y, and Position_Z. In this process, the speed of the robotic arm's movement is determined by the hyper-parameters in the configuration file.

In the real material transfer system, the number of material input and output ports is limited, and the robotic arm only stops at the material input and output ports, leading to its limited system state. If, during the simulation execution, once the displacement information of the robotic arm from input port A to output port B is calculated, it is recorded in a transition table, the next time it goes from A to B, there is no need to recalculate the displacement information, which will further improve the simulation speed. Based on this observation, we propose state-aware simulation acceleration.

The adoption of state-aware simulation acceleration is feasible due to the inherent characteristics of real industrial control models, especially those with limited system states, in which the behavior of the system is constrained by the limited range of inputs and possible states. Thus, during simulation, changes in the system state can be dynamically recorded and corresponded with inputs and outputs. This record facilitates the effective identification of recurring system states and inputs in subsequent simulation steps. Therefore, if the current system state and input match those previously recorded, there is no need to re-execute that simulation step. The system can transit directly to the next state and produce the corresponding output.

To implement state-aware simulation acceleration, it is crucial to first delineate the constituents of the system state within a Simulink model, as well as system input and output. To enhance clarity in the following descriptions, we formally define them as follows:

Definition 1 (System State): $SS = \{P_1, \ldots, P_m, G_1, \ldots, G_n\}$, where P_j $(1 \leq j \leq m)$ represents a model hyperparameter and each G_k $(1 \leq k \leq n)$ represents a global variable. Throughout the simulation process, the model can be conceptualized as a function, where each simulation step equates to executing this function. Ideally, the behavior of a function should be deterministic, meaning the same inputs yield the same outputs. However, if the function's internal dynamics are contingent upon external variables, it may affect its behavior. In the context of a model, these external variables consist of model parameters and global variables.

Definition 2 (Test Case): $TC = \{I_1, I_2, \ldots, I_n\}$, denotes the numerical values assigned to all input ports of the system, where each I_j $(1 \leq j \leq n)$ corresponds to a specific input port. A test case embodies a distinct combination of input values applied to the model during simulation, delineating a particular scenario or condition for evaluating the model's behavior. It's noteworthy that one TC stands for the test case of a single simulation step.

Definition 3 (System Output): $SO = \{O_1, \ldots, O_n, D\}$, where each O_j $(1 \leq j \leq n)$ represents one of the system's output port value, and D represents the diagnostic information. System outputs consist of the observed values or responses from the model's output ports, including any diagnostic information generated during simulation to aid in analysis and troubleshooting. As the skipped simulation executions are identical to those previously performed, the outputs and diagnostic information must be recorded for subsequent use. However, since repeated executions do not alter the coverage rate, SO does not need to include coverage information.

State & I/O Collection: When implementing state-aware simulation acceleration, we first need to identify and collect the model parameters and global variables that constitute the system state. Model hyperparameters are typically fixed configuration values, while global variables are quantities that change during the simulation process. Together, they determine the state of the model system at any given time. We need to extract and record the values of all these hyperparameters and variables as part of the system state $SS(P, G)$.

At the same time, we need to obtain the values of the model's input ports as test case inputs $TC(I_1, I_2, \ldots)$. The input port values represent the input data of the model from external systems during runtime. Together with the changes in

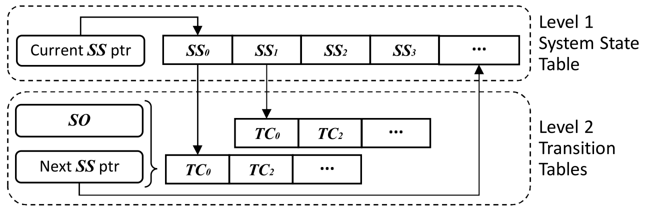

Current *SS* ptr
*SS*₀ *SS*₁ *SS*₂ *SS*₃ ···
Level 1 System State Table
SO
Next *SS* ptr
*TC*₀ *TC*₂ ···
*TC*₀ *TC*₂ ···
Level 2 Transition Tables

Fig. 6. State transition hash table.

the internal state, they determine the execution behavior of the model. Therefore, at the beginning of each simulation step, the corresponding input values need to be retrieved from the provided test cases as the input data for the current step.

The collected system state *SS* and input *TC* will be used to determine whether the current simulation step can be skipped. In the subsequent state transition process, we will check if there are any previously recorded state transition entries that match the current state *SS* and input *TC*. If a match is found, the execution of the current step can be skipped directly, thereby accelerating the simulation. Otherwise, the current step needs to be executed, and the newly generated state transition entry will be recorded for subsequent use.

System State Transition: In the design of State-Aware Acceleration, the System State Transition plays a crucial role. As illustrated in Fig. 6, AccSiM employs a two-level hash table data structure to store and retrieve information related to system state transitions. This structure consists of a system state table and a transition table. Each item of the first-level system state table has two fields, i.e., `state` and `pointer`, where the `state` field stores a *SS* and the `pointer` field records the pointer to its second-level transition table. The transition table uses *TC* as keys, with corresponding values being *SO* and its next *SS* pointer to the system state table.

Additionally, the system transition hash table maintains a global pointer to the system state table, pointing the current *SS*. This two-level hash table structure enables efficient storage and retrieval of system state transition information. The first-level system state table rapidly locates the second-level input transition table associated with the current *SS*, while the second-level input transition table stores and retrieves the *SO* and next *SS* pointer corresponding to the (*SS*, *TC*) pair. In the simulation startup phase, it is necessary to initialize the System State Transition Hash Table. First, obtain the system's initial SS_{init}, store it into `SystemStateTable[hash(`SS_{init}`)].state`, and set `SystemStateTable[hash(`SS_{init}`)].pointer` as `NULL`. And then set the current *SS* pointer at `SystemStateTable[hash(`SS_{init}`)]`.

Fig. 7 illustrates the workflow of AccSiM in "one-step" simulation with state-aware acceleration. While importing *TC* into the simulation system, the first step involves querying the state transition hash table to ascertain if there exists a corresponding transition entry correlating with this *TC* under current *SS*. If a matching record is found, the execution of this simulation step shall be skipped. Meanwhile, the corresponding output information for that simulation step (including the new *SS* and *SO*) is retrieved from the state

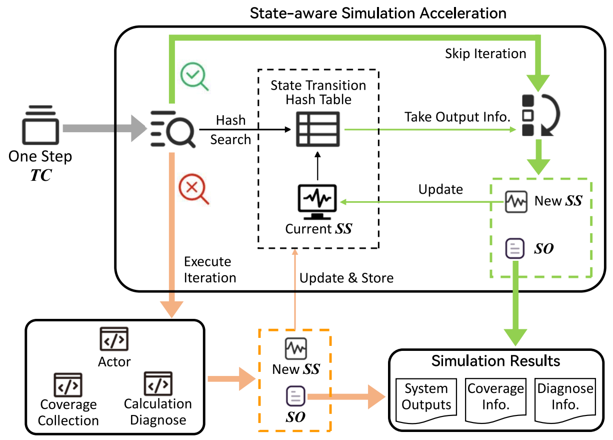


Fig. 7. Workflow of model simulation with state-aware method.

transition hash table. Subsequently, the current *SS* within State-aware Simulation Acceleration (Current *SS* ptr) are adjusted based on the newly acquired system state (Next *SS* ptr), and the outcome of this simulation step is delineated.

Conversely, in the absence of a corresponding record within the state transition hash table, the system code proceeds with its usual execution. Upon completion of this execution, the newly attained *SS* and *SO* are duly updated within the state transition hash table, thus ensuring their availability for subsequent use. The update process includes storing new *TC* and its corresponding *SO* into Transition Table, inserting the new *SS* into System State Table while setting Next *SS* ptr and Current *SS* ptr at it. Through this approach, AccSiM effectively leverages previously computed state transition information, avoiding redundant execution of identical simulation computations and significantly accelerating simulation process.

However, when the redundant step rate is minimal, the state-aware acceleration approach might result in substantial memory and time overhead. To mitigate this problem, we implement a heuristic algorithm with two key configuration parameters: `FIRST_X_STEPS` and `REDUNDANT_RATE`. If the redundant step rate is below `REDUNDANT_RATE` (default 50%) within the first `FIRST_X_STEPS` (default 10 000) simulation steps, the state-aware acceleration approach will be automatically disabled, and the simulation progress will return to standard code execution. Moreover, if users are already familiar with the characteristics of the model, they can manually configure the state-aware method by toggling its settings, allowing further customization and optimization for specific use cases.

C. 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, AccSiM needs to configure such actor information to obtain

① Code of main function

```
 1 int main(int argc, char* argv[]) {
 2   TestCase_Init(); Model_Init(); State_Init();
 3   // Simulation Loop of model
 4   for(int step = 0; step < TOTAL_STEP; step++) {
 5     int Inport_A = takeTestCase(); int Inport_B = takeTestCase();
 6     int Outport;
 7     if(MatchSkip(Inport_A, Inport_B, &GlobalVariables)){
 8       SkipIteration(&Outport, &GlobalVariables); }
 9     else{
10       Model_Exe(Inport_A, Inport_B, &Outport);
11       RecordNewExe(Inport_A, Inport_B, &Outport, &GlobalVariables); }
12   }
13   outputResult();
14 }
```

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");
7   diagnose_Model_Minus(Minus_Out, Inport_A, Inport_B);
8   ...
```

Fig. 8. Sample of final simulation code.

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, AccSim encapsulates it within a model system function, exemplified in the second part of Fig. 8. Then AccSim generates a main function to implement the simulation loop, where the model system function (line 10) is invoked to carry out the simulation process. The state-aware acceleration method can be optionally inserted in to the main function.

An illustrative example of a main function is shown in the first part of Fig. 8. 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 (line 5), as well as initializing the state aware acceleration method (line 2). Lines 7 to 11 illustrate the execution process of state-aware acceleration. Moreover, the code responsible for outputting simulation results (including diagnostic and coverage information) is placed at the end of the main function (line 13).

IV. Implementation

AccSim[1] is implemented in C++ with 41 536 lines of code. During the model preprocessing phase, the Simulink model, stored in XML format, is parsed using the TinyXML library. This approach enables the efficient extraction of actor information, which is essential for generating both instrumentation code and actor code. 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.

[1]The implementation of AccSim and the benchmark models are available on the following website: https://github.com/ClarkCC36/AccSiM.

TABLE II
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

In the State Transition Hash Table shown in Fig. 6, we use the CRC64 hashing method. The first level, namely the System State Table, consists of a hash table with a size of 10 007. Each state in this table corresponds to a second-level Transition Table, which has a size of 101. To handle hash collisions, we apply the chaining method.

V. Evaluation

A. Experiment Setup

To evaluate AccSim, we conducted a comparative analysis against SSE with 10 Simulink benchmark models. As shown in Table II, all benchmark models are derived from industry and deployed in embedded scenarios. All experiments were executed in a consistent environment (Windows 11, Intel i7-13700F CPU, 32GB RAM). The simulation code was compiled by a C/C++ Compiler (GCC 8.1.0), employing -O3 optimization flag. Each data point of these experiments represents the average of five experiment runs, ensuring the reliability and stability of the results. Our experiments are divided into five main parts: simulation time, capability of error diagnosis, effectiveness of coverage collection, memory overhead and performance overhead of instrumentation. Note that for comparison on error diagnosis and coverage collection, we solely compared AccSim with SSE, as SSE_{ac} and SSE_{rac} cannot perform these simulation functions.

Simulation Time: In this experiment, we first measured the translation time of the benchmark models, which is the duration required to convert the model file into its generated code. We also measured the compilation time, which is the time taken to convert the final simulation code into an executable file. Together, these times are combined as the initialization time of AccSim. In contrast, we measured the initialization times of SSE, SSE_{ac}, and SSE_{rac}. Since interpreted execution is used by SSE and SSE_{ac}, they do not involve translation and compilation processes. However, both undergo an initialization phase before entering the simulation loop. During this phase, Simulink merges library actors into the model, determines signal widths, data types, and sample times, calculates actor parameters, establishes the execution order of actors, and allocates memory. SSE_{rac}, on the other hand, also converts the model into code for simulation, and

TABLE III
BENCHMARK MODELS WITH MANUALLY INJECTED ERRORS

Model	#Error	Error Types
CPUT	2	Overflow, Downcast
CSEV	2	Overflow, Overflow
FMTM	3	Overflow, Array out of bounds, Downcast
LANS	3	Devided by zero, Downcast, Array out of bounds
LEDLC	2	Overflow, Downcast
RAC	2	Downcast, Devided by zero
SPV	2	Overflow, Downcast
TCP	3	Overflow, Array out of bounds, Downcast
TWC	2	Overflow, Devided by zero
UTPC	2	Downcast, Devided by zero

TABLE IV
TIME CONSUMPTION OF SIMULATION INITIALIZATION PHASE FOR ACCSIM AND SSE (UNIT: SECOND)

Model	Translate Time	Compile Time	Initialzation Time of ACCSIM	Initialization time of SSE: SSE	SSE_{ac}	SSE_{rac}
CPUT	0.19	0.54	0.73	6.66	6.83	16.49
CSEV	0.09	0.42	0.51	9.35	9.11	20.42
FMTM	0.30	0.77	1.07	6.69	6.73	15.71
LANS	0.30	0.73	1.03	8.20	8.45	19.31
LEDLC	0.18	0.60	0.78	5.19	5.31	12.83
RAC	0.49	1.02	1.50	8.45	8.81	19.6
SPV	0.10	0.37	0.47	6.87	6.84	16.09
TCP	0.28	0.78	1.06	6.05	6.05	15.22
TWC	0.12	0.32	0.45	7.81	7.66	18.82
UTPC	0.20	0.45	0.65	6.18	6.44	14.83
Average	0.22	0.60	0.82	7.15	7.22	16.93

its initialization phase includes model translation process and code compilation process.

After that, we compared the simulation time differences across benchmarks for ACCSIM, $\text{ACCSIM}_{\text{sa}}$ (with state-aware acceleration), SSE, SSE_{ac}, and SSE_{rac}. The comparison for each benchmark was conducted using the same test cases, which are extracted from the input sequences of the benchmark in real industrial scenarios. To meet the industrial requirements of long-term execution and stability tests, the simulations were conducted with a significant step size of 50 million.

Capability of Error Diagnosis: We first manually injected several errors into each benchmark model. The number and types of inserted errors are shown in Table III. We then simulated these benchmarks using ACCSIM, $\text{ACCSIM}_{\text{sa}}$ and SSE, recording whether they could detect the errors and the time differences in detecting the same errors.

Effectiveness of Coverage Collection: Since coverage collection typically relies on randomly generated test cases, which does not align with the intended design scenario of the State-Aware approach, where input variations are minimal, we conducted the coverage collection experiments just between ACCSIM and SSE, with the same test cases generated through a random approach. The evaluation specifically centered on comparing the coverage achieved by all methodologies within a consistent simulation time frame. Coverage metrics, including actor, condition, decision, and MC/DC, were systematically recorded at simulation intervals of 5 s, 15 s, and 60 s.

Memory Overhead: In addition to comparing simulation efficiency and functionality, we also conducted experiments on memory overhead. Since the memory overhead of SSE is mixed with the entire MATLAB process, it is difficult to measure the memory overhead of SSE separately. We compared the memory usage between ACCSIM and $\text{ACCSIM}_{\text{sa}}$. The measurement method involved observing the Windows Task Manager during the simulation program's execution and recording the maximum memory usage for each experiment.

Performance Overhead of Instrumentation: In contrast to conventional control models, ACCSIM introduces additional data diagnostic instrumentation in models involving complex numerical computations. This instrumentation adds extra conditional logic, thereby introducing additional performance overhead. To assess the additional performance overhead introduced by implementing simulation function instrumentation, we commented out all instrumented code to obtain the noninstrumented C++ code. Subsequently, we compiled this code segment and recorded its execution time. Furthermore, we add four publicly available models (Simpson, ABS, Filter_high, Filter_low) to evaluate the performance overhead for pure computational models.

B. Evaluation on Simulation Time

The results presented in Table IV indicate that the average time consumption of initialization process for ACCSIM is only 0.82s, making it 8.7×, 8.8×, 20.5× faster than SSE, SSE_{ac} and SSE_{rac} respectively. The translation time ranges from 0.09s for CSEV to 0.49s for RAC. This suggests that translation time tends to increase with the number of actors in a model, as more components require parsing and processing during this phase. The compilation time ranges from 0.32s for TWC to 1.02s for RAC. The compilation time appears to correlate more closely with the number of subsystems, as ACCSIM generates separate code files for each subsystem, which increases the workload during the compilation process.

ACCSIM achieves significant performance improvement on simulation efficiency as shown in Table V. Compared to SSE, SSE_{ac} and SSE_{rac}, ACCSIM displayed an average efficiency improvement of 215.3×, 76.3× and 19.8×, respectively. Additionally, the state-aware acceleration method achieved an average speedup of 2.8× compared to ACCSIM, resulting in simulation efficiency improvements of 568.4×, 200.3×, and 49.5× over SSE, SSE_{ac}, and SSE_{rac}, respectively.

We observe that the acceleration ratios of four models, namely LANS, LEDLC, SPV, 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 ACCSIM 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,

TABLE V
COMPARISON OF SIMULATION TIME (UNIT: SECOND)

Model	AccSim	AccSim$_{sa}$	SSE	SSE$_{ac}$	SSE$_{rac}$
CPUT	4.21	1.35	167.67	69.55	37.41
CSEV	0.77	0.58	75.06	43.97	35.58
FMTM	2.42	0.95	70.61	58.31	32.80
LANS	3.61	1.12	1603.21	536.81	99.96
LEDLC	4.31	1.50	1688.20	512.75	48.66
RAC	3.45	1.21	108.99	70.77	48.35
SPV	1.67	0.80	934.88	375.66	34.60
TCP	2.09	1.02	768.05	158.26	46.15
TWC	2.05	0.82	182.27	76.22	41.34
UTPC	10.88	1.98	1120.77	430.06	140.38

which includes conditional statements, is less amenable to such optimizations by compilers or processors. Consequently, higher acceleration ratios are achieved in these models.

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, AccSim 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.

Through a comprehensive analysis of the simulation data from AccSim$_{sa}$, we found that the number of distinct states in these models is relatively limited, typically ranging from a few dozen to over 3000. Furthermore, the execution process is predominantly concentrated on a small subset of these states. For example, in the LEDLC model, the entire operation spans 41 states, yet over three-fourths of the execution time is focused on just 18 core states. In contrast, we conducted additional experiments using a set of pure computational models with very few redundant loops (174 on average) to evaluate the time overhead of state-aware approach. With the default values for `FIRST_X_STEPS` and `REDUNDANT_RATE`, AccSim$_{sa}$ completed the simulation in 3.01 s on average, while AccSim took 2.86 s, resulting in an additional time overhead of only 5.2%. This demonstrates that, in scenarios with minimal redundant step rate, the overhead introduced by AccSim$_{sa}$ can be effectively controlled by our heuristic algorithm.

C. *Capability of Error Diagnosis*

Detailed results are presented in Table VI. We observed that all injected errors can be detected by AccSim, AccSim$_{sa}$ and SSE, with AccSim detecting the errors on average 236.2× faster than SSE. This indicates that AccSim not only achieves error detection during simulation but also operates with much higher efficiency compared to SSE. Additionally, we observed that AccSim$_{sa}$ achieves an average speedup of 1.4× compared to AccSim in error detection speed.

TABLE VI
COMPARISON OF ERROR DIAGNOSE TIME (UNIT: SECOND)

Model	Error Types	Discover Time		
		AccSim	AccSim$_{sa}$	SSE
CPUT	Overflow	2.43	1.09	112.22
	Downcast	3.56	2.67	157.36
CSEV	Overflow	0.74	0.35	450.1
	Devided by zero	0.91	0.62	482.3
FMTM	Overflow	1.97	1.52	51.03
	Array out of bounds	2.02	1.89	60.35
	Downcast	2.28	2.01	69.40
LANS	Devided by zero	2.17	1.68	987.39
	Downcast	3.11	2.84	1405.35
	Array out of bounds	3.52	3.05	1587.52
LEDLC	Overflow	3.02	2.51	1235.68
	Downcast	3.93	3.01	1603.27
RAC	Downcast	2.42	1.82	80.38
	Devided by zero	2.95	2.15	98.32
SPV	Overflow	1.21	0.83	633.49
	Downcast	1.01	0.72	647.89
TCP	Overflow	1.39	0.98	602.80
	Array out of bounds	1.68	1.12	617.18
	Downcast	1.92	1.19	732.71
TWC	Overflow	1.23	0.82	134.74
	Devided by zero	1.46	1.02	142.74
UTPC	Overflow	7.89	4.22	867.28
	Devided by zero	8.63	4.96	902.48

Take the CSEV benchmark model as a case study. 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, AccSim 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 overflow 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 overflow error. To identify this error, AccSim employs the `sizeof()` function to determine the data sizes of both the inputs and outputs in the product calculation. A wrap on overflow error is indicated if these sizes do not align.

TABLE VII
COVERAGE OF ACCSIM AND SSE

Model	Time	Actor		Condition		Decision		MC/DC	
	(s)	ACCSIM	SSE	ACCSIM	SSE	ACCSIM	SSE	ACCSIM	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%

The first wrap on overflow is detected by ACCSIM 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 ACCSIM.

D. Effectiveness of Coverage Collection

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. Detailed results are presented in Table VII.

Our experiments indicate that within just 5 s, all 4 coverage metrics achieved by ACCSIM surpass 60 s of SSE simulation, for all models apart from the TCP model. As for TCP, after a very brief 15-s simulation, its coverage comprehensively surpassed the results obtained through simulation on SSE. ACCSIM demonstrates significant efficiency improvement in coverage collection.

E. Memory Overhead

Based on the experimental results shown in Table VIII, It is observed that ACCSIM$_{\text{sa}}$ incurs significant memory usage that is approximately 2.95× that of ACCSIM across all benchmark models. This increase primarily stems from the state-aware acceleration method requiring additional storage for state information and transition tables. Furthermore, as the number of states increases, there is a corresponding rise in memory consumption observed during the experiments.

TABLE VIII
MEMORY CONSUMPTION OF ACCSIM AND ACCSIM$_{\text{SA}}$ (UNIT: MB)

Model	ACCSIM	ACCSIM$_{\text{sa}}$	Ratio
CPUT	605.8	2010.3	3.32×
CSEV	620.2	2037.4	3.29×
FMTM	606.1	1967.3	3.25×
LANS	987.3	2416.8	2.45×
LEDLC	207.7	730.8	3.52×
RAC	987.3	2189.0	2.22×
SPV	446.1	1782.7	3.99×
TCP	415.1	1082.3	2.61×
TWC	813.2	1971.9	2.42×
UTPC	431.7	1039.6	2.41×

In contrast, ACCSIM maintains consistent memory overhead across multiple experimental runs. Analysis of the simulation code generated by ACCSIM reveals that memory consumption primarily originates from the Actor Info Collection, which utilizes fixed-length data structures to record component output values, resulting in predictable memory usage.

Besides, Although ACCSIM$_{\text{sa}}$ brings about significant performance improvements, these gains come at the cost of increased memory consumption. Therefore, when choosing between ACCSIM and ACCSIM$_{\text{sa}}$, users need to balance the tradeoff between performance enhancement and memory usage, especially in scenarios involving large and complex models or systems with memory constraints. This tradeoff underscores the importance of selecting an appropriate simulation method based on specific hardware environments and model characteristics.

F. Performance Overhead of Instrumentation

The experimental results are presented in Table IX. Following the benchmarks, the subsequent entries (Simpson, ABS, Filter_high, Filter_low) denote four publicly available purely computational models. It can be observed that in the benchmark scenario, instrumentation by ACCSIM incurred an average overhead of 54.7%, maintaining an average speedup of 209.3× compared to SSE. However, for the purely computational models, ACCSIM introduces an average performance overhead of 279.5% compared to noninstrumented C++ code (ACCSIM without instrumentation). Nevertheless, it maintains a substantial acceleration ratio of 217.9× comparing to SSE.

Be aware that there is an anomaly in the instrumentation overhead for CPUT and LANS as indicated in the table. The ACCSIM execution time of these two models is unexpectedly faster compared to the code without instrumentation.To understand this phenomenon, we conducted an analysis based on the assembly code of corresponding C++ code shown in Fig. 9.

In this code, line 12 represents instrumentation code for bounds checking to detect array out-of-bounds access. In our experiments, when we commented out this line, the execution time was slower than when it was retained. Upon analyzing the assembly code, we observed that without the instrumentation code (commenting out line 12), the compiler translated the code from lines 5 to 10 into repeated conditional move instructions (cmove). Essentially, it integrated the if statement from line 5 into the assignment statement at line 9

TABLE IX
PERFORMANCE OVERHEAD OF SIMULATION FUNCTIONALITY INSTRUMENTATION (UNIT: SECOND)

Model	**AccSim**	**AccSim without Instrumentation**	**Overhead of Instrumentation**	**Improvement (vs. SSE)**
CPUT	4.21	7.06	-40.4%	39.8×
CSEV	2.03	1.77	14.7%	37.0×
FMTM	2.42	1.47	64.6%	29.1×
LANS	3.61	3.64	-0.8%	444.1×
LEDLC	4.31	2.23	93.3%	391.7×
RAC	3.45	2.63	31.2%	31.6×
SPV	1.67	1.12	49.1%	559.8×
TCP	2.09	0.89	134.8%	368.0×
TWC	2.05	0.70	192.9%	88.7×
UTPC	10.88	8.06	35.0%	103.0×
Simpson	2.86	0.91	214.3%	137.8×
ABS	2.64	0.93	183.9%	73.2×
Filter_high	1.66	0.35	374.3%	336.9×
Filter_low	1.47	0.33	345.5%	323.6×

```
 1 ...
 2 Assignment_init = 0;
 3 for (int For_Iterator_Out1 = 0; For_Iterator_Out1 < 16; For_Iterator_Out1++) {
 4   ...
 5   if (Assignment_init == 0) {
 6     Assignment_init = 1;
 7     int batchIndex;
 8     for (batchIndex = 0; batchIndex < 16; batchIndex++)
 9       Assignment_Outport1[batchIndex] = Data_StoreRead1_Outport1[batchIndex];
10   }
11   Assignment_Outport1[For_Iterator_Out1] = Switch_Outport1;
12   diagnoseCalculator_system_85_Assignment(...);
13   ...
14 }
```

Fig. 9. Part of simulation code which CPUT and LANS share.

and performed loop unrolling 16 times. This resulted in the execution of lines 6 to 9, originally meant to be executed once, being repeated 16 times. However, with the insertion of line 12, the aforementioned code was not translated into cmove instructions by the compiler. Instead, it was translated into normal conditional jump instructions. As modern processors often have branch prediction capabilities, the code using conditional jump instructions is significantly faster compared to the repetitive cmove instructions.

VI. DISCUSSION

Threats to Validity: At present, AccSim mainly supports code-based simulation for discrete models, but a key limitation of AccSim is its current lack of capability in supporting continuous models. In contrast to discrete models, which experience changes at specific intervals, continuous models are fundamental in embedded systems for their tight interaction with the environment in terms of sensing/actuation and communication [30], [31], [32]. Expanding AccSim to encompass both discrete and continuous models would significantly enhance its versatility and utility. To enhance the simulation of continuous models through code-based approaches, AccSim could either transform analog descriptions into C++-based languages [32] or incorporate numerical solvers such as the Adams solver [33] to efficiently address the differential equations inherent in continuous models.

Extensibility of AccSim*:* AccSim 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 [5], [8], [34]. To support code-based simulation for these tools, AccSim must be capable of parsing their unique model representations while generating the corresponding code. One possible strategy is to build a well-structured intermediate representation (IR) that ensures compatibility with various model-driven design tools [24], [35]. Additionally, to further enhance simulation efficiency, AccSim could explore leveraging optimization techniques used by other code generators [36], [37].

Limitations of AccSim$_{sa}$*:* As previously described, when the number of combinations of values for global variables and parameters in the model is limited, and the number of combinations of input parameters is also not large, the number of internal states in the model can be small, leading to a majority of the simulation steps involving redundant computations. By using state-Aware approach to avoid these redundant computations, simulation efficiency can be significantly enhanced without any loss of simulation functionality.

However, when the number of model states or the number of combinations of input parameter values is large, the hash table required for implementing the State-Aware function will consume a considerable amount of memory. In such cases, the proportion of redundant computation steps decreases significantly, and the time cost associated with determining whether a computation is redundant can offset the performance benefits gained from avoiding redundant computations. This results in a reduced increase in overall simulation efficiency and, in some cases, may even lead to a decrease in simulation efficiency. Consequently, the State-Aware approach is mainly suitable for industrial models with a relatively small number of states and input parameter combinations. In other scenarios, the overhead introduced can be effectively controlled by our heuristic algorithm. Furthermore, the State-Aware approach can be manually disabled as an optional feature.

VII. RELATED WORKS

Model-Driven Design and Simulink: Model-driven design is a software development method that has been widely used in safety-critical embedded scenarios [38], [39]. 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.

Both [32] and [35] focus on enhancing the simulation speed of heterogeneous systems by shifting complexity from runtime to generation time. Reference [32] automates the conversion of analog models into C++ for easier integration with virtual platforms, facilitating the joint simulation of digital and analog components in smart systems. Reference [35] uses an IR, HIF, to unify models from various tools and languages, generating a homogeneous C++ event-driven simulator for the initial heterogeneous models.

The main approach of [32] and [35] to enhance simulation speed is to reduce the details of each model that must be simulated to the minimum necessary. In contrast, AccSim accelerates simulation by utilizing binary execution as a replacement for SSE's interpretive execution. Furthermore, AccSim focuses on Simulink's data flow and control flow models. Meanwhile, [32] and [35] concentrate on the co-simulation of heterogeneous hardware and software designs within smart systems.

VIII. Conclusion

In this article, we have presented AccSim, a novel approach to accelerate model simulation through automated code generation, directly addressing the aforementioned challenges. To overcome the challenge of identifying essential simulation data, AccSim employs a preprocessing step to extract key actor information and signal details, efficiently filtering the data for simulation analysis. In tackling the challenge of analyzing the acquired data, AccSim generates instrumentation code for simulation functionalities, including error diagnosis and coverage statistics. To address the challenge of skipping duplicate iterations, AccSim incorporates state-aware acceleration approach, optimizing the simulation process by skipping redundant loops and reducing unnecessary computational overhead. Finally, AccSim synthesizes the final code by integrating instrumentation code with actor code generated from templates, alongside test case import code. Through this code-based simulation, AccSim rapidly produces results containing coverage and diagnostic information.

We implemented AccSim and evaluated it on several benchmark Simulink models. The results demonstrate that compared to SSE, AccSim 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. Furthermore, through the state-aware acceleration method, AccSim yielded an additional 2.8× speedup. In the future, we plan to extend AccSim's capabilities for supporting continuous models and other modeling environments, and expand the application scope of the state-aware approach.

References

[1] Y. Cheng, Z. Yu, Z. Su, T. Chen, X. Zhang, and Y. Jiang, "AccMoS: Accelerating model simulation for simulink via code generation," in *Proc. 61st ACM/IEEE Design Autom. Conf.*, 2024, pp. 1–6.

[2] F. Pasic, "Model-driven development of condition monitoring software," in *Proc. 21st ACM/IEEE Int. Conf. Model Driven Eng. Lang. Syst. Companion*, 2018, pp. 162–167.

[3] F. Balarin et al., *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*, vol. 404. New York, NY, USA: Kluwer Academic Publ., 2012.

[4] G. Berry, "Circuit design and verication with Esterel v7 and Esterel studio," in *Proc. IEEE Int. High Level Design Validation Test Workshop*, 2007, pp. 133–136.

[5] Y. Jiang et al., "Tsmart-Galsblock: A toolkit for modeling, validation, and synthesis of multi-clocked embedded systems," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 711–714.

[6] L. Berzi et al., "Brake blending strategy on electric vehicle co-simulation between MATLAB simulink® and simcenter amesim™," in *Proc. IEEE 5th Int. Forum Res. Technol. Soc. Ind. (RTSI)*, 2019, pp. 308–313.

[7] "Simulink documentation." Simulink and MATLAB. 2023. [Online]. Available: https://www.mathworks.com/help/simulink/index.html

[8] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," in *Readings in Hardware/Software Co-Design* (Systems on Silicon), G. De Micheli, R. Ernst, and W. Wolf, Eds. San Francisco, CA, USA: Morgan Kaufmann, 2002, pp. 527–543.

[9] H. Liu, X. Liu, and E. A. Lee, "Modeling distributed hybrid systems in ptolemy ii," in *Proc. Amer. Control Conf.*, vol. 6, 2001, pp. 4984–4985.

[10] J. Eker et al., "Taming heterogeneity—The ptolemy approach," *Proc. IEEE*, vol. 91, no. 1, pp. 127–144, Jan. 2003.

[11] D. Hästbacka, T. Vepsäläinen, and S. Kuikka, "Model-driven development of industrial process control applications," *J. Syst. Softw.*, vol. 84, no. 7, pp. 1100–1113, 2011.

[12] K. Thramboulidis, D. Perdikis, and S. Kantas, "Model driven development of distributed control applications," *Int. J. Adv. Manuf. Technol.*, vol. 33, pp. 233–242, Jun. 2007.

[13] G. Trombetti et al., "An integrated model-driven development environment for composing and validating distributed real-time and embedded systems," in *Model-Driven Software Development*. Heidelberg, Germany: Springer, 2005, pp. 329–361.

[14] T. Z. Asici, B. Karaduman, R. Eslampanah, M. Challenger, J. Denil, and H. Vangheluwe, "Applying model driven engineering techniques to the development of contiki-based IoT systems," in *Proc. IEEE/ACM 1st Int. Workshop Softw. Eng. Res. Pract. Internet Things (SERP4IoT)*, 2019, pp. 25–32.

[15] C. Zhao, H. Yan, D. Liu, H. Zhu, Y. Wang, and Y. Chen, "Co-simulation research and application for active distribution network based on ptolemy ii and simulink," in *Proc. China Int. Conf. Electricity Distrib. (CICED)*, 2014, pp. 1230–1235.

[16] K. Jahed and J. Dingel, "Enabling model-driven software development tools for the Internet of Things," in *Proc. IEEE/ACM 11th Int. Workshop Model. Softw. Eng. (MiSE)*, 2019, pp. 93–99.

[17] F. Rademacher, J. Sorgalla, S. Sachweh, and A. Zündorf, "A model-driven workflow for distributed microservice development," in *Proc. 34th ACM/SIGAPP Symp. Appl. Comput.*, 2019, pp. 1260–1262.

[18] S. Bairami, M. Salimi, and D. Mirabbasi, "A novel method for maximum power point tracking of the grid-connected three-phase solar systems based on the PV current prediction," *Chin. J. Electron.*, vol. 32, no. 2, pp. 353–364, 2023.

[19] E.-J. Wagenmakers, P. Grünwald, and M. Steyvers, "Accumulative prediction error and the selection of time series models," *J. Math. Psychol.*, vol. 50, no. 2, pp. 149–166, 2006.

[20] F. Zhang, H. Stähle, G. Chen, C. C. C. Simon, C. Buckl, and A. Knoll, "A sensor fusion approach for localization with cumulative error elimination," in *Proc. IEEE Int. Conf. Multisensor Fusion Integration Intell. Syst. (MFI)*, 2012, pp. 1–6.

[21] K. Thramboulidis and G. Frey, "An MDD process for IEC 61131-based industrial automation systems," in *Proc. ETFA*, 2011, pp. 1–8.

[22] A. Hamed, M. Safar, M. W. El-Kharashi, and A. Salem, "AUTOSAR-based communication coprocessor for automotive ECUS," in *Proc. Design, Autom. Test Europe Conf. Exhibition (DATE)*, 2016, pp. 1026–1027.

[23] J. A. de la Puente, J. Garrido, J. Zamorano, and A. Alonso, "Model-driven design of real-time software for an experimental satellite," *IFAC Proc. Vol.*, vol. 47, no. 3, pp. 1592–1598, 2014.

[24] Z. Su et al., "MDD: A unified model-driven design framework for embedded control software," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 10, pp. 3252–3265, Oct. 2022.

[25] K. Balasubramanian, A. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema, "Developing applications using model-driven design environments," *Computer*, vol. 39, no. 2, pp. 33–40, Feb. 2006.

[26] X. Zhang, P. Wang, J. Lin, H. Chen, J. Hong, and L. Zhang, "Real-time nonlinear predictive controller design for drive-by-wire vehicle lateral stability with dynamic boundary conditions," *Fund. Res.*, vol. 2, no. 1, pp. 131–143, 2022.

[27] I. Rahul and R. Hariharan, "Enhancement of solar PV panel efficiency using double integral sliding mode MPPT control," *Tsinghua Sci. Technol.*, vol. 29, no. 1, pp. 271–283, 2023.

[28] "Simulink model coverage document." Simulink and MATLAB. 2023. [Online]. Available: https://www.mathworks.com/help/slcoverage/ug/model-coverage.html

[29] Z. Su et al., "Code synthesis for dataflow-based embedded software design," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 1, pp. 49–61, Jan. 2022.

[30] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann, "Automated test suite generation for time-continuous simulink models," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 595–606.

[31] W. M. Hanemann, "Discrete/continuous models of consumer demand," *Econometrica J. Econ. Soc.*, vol. 52, no. 3, pp. 541–561, 1984.

[32] M. Lora, S. Vinco, E. Fraccaroli, D. Quaglia, and F. Fummi, "Analog models manipulation for effective integration in smart system virtual platforms," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 2, pp. 378–391, Feb. 2018.

[33] T. Brezina, Z. Hadas, and J. Vetiska, "Using of co-simulation adams-simulink for development of mechatronic systems," in *Proc. 14th Int. Conf. Mechatronika*, 2011, pp. 59–64.

[34] G. Berry, "SCADE: Synchronous design and validation of embedded control software," in *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*. Dordrecht, The Netherlands: Springer, 2007, pp. 19–33.

[35] M. Lora, S. Vinco, and F. Fummi, "Translation, abstraction and integration for effective smart system design," *IEEE Trans. Comput.*, vol. 68, no. 10, pp. 1525–1538, Oct. 2019.

[36] Z. Su et al., "HCG: Optimizing embedded code generation of simulink with SIMD instruction synthesis," in *Proc. 59th ACM/IEEE Design Autom. Conf.*, 2022, pp. 1033–1038.

[37] Z. Yu et al., "Mercury: Instruction pipeline aware code generation for simulink models," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 11, pp. 4504–4515, Nov. 2022.

[38] P. H. Feiler, "Model-based validation of safety-critical embedded systems," in *Proc. IEEE Aerosp. Conf.*, 2010, pp. 1–10.

[39] H. Hansson, M. Åkerholm, I. Crnkovic, and M. Torngren, "SaveCCM—A component model for safety-critical real-time systems," in *Proc. 30th Euromicro Conf.*, 2004, pp. 627–635.

Yifan Cheng received the B.S. degree in computer science from the University of Electronic Science and Technology of China, Chendgu, China, in 2022, where he is currently pursuing the Ph.D. degree in cybersecurity.

His research interests are in the areas of model driven development, embedded software engineering, and software security.

Zehong Yu received the B.S. degree in software engineering from Southeast University, Nanjing, China, in 2021, and the M.S.E. degree in software engineering from Tsinghua University, Beijing, China, in 2024, where he is currently pursuing the Ph.D. degree in software engineering.

His research interests are in the areas of model driven development and embedded software engineering.

Zhuo Su received the B.S. degree in software engineering from Northeastern University, Shenyang, China, in 2018, and the Ph.D. degree in software engineering from Tsinghua University, Beijing, China, in 2023.

He is currently a Postdoctoral Fellow with the School of Software, Tsinghua University. His research interests are in the areas of model driven development and embedded software engineering.

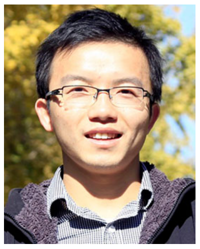
Ting Chen (Member, IEEE) received the Ph.D. degree from the University of Electronic Science and Technology of China (UESTC), Chendgu, China, in 2013.

He is currently a Professor with the School of Computer Science and Engineering, UESTC. He has authored or co-authored tens of high-quality papers in prestigious conferences and journals. His research interests are in the areas of blockchain, smart contract, and software security.

Prof. Chen's work was the recipient of several best paper awards, including the INFOCOM 2018 Best Paper Award.

Xiaosong Zhang received the M.S. and Ph.D. degrees in computer science from the University of Electronic Science and Technology of China, Chengdu, China, in 1999 and 2011, respectively.

He is currently a Professor with the University of Electronic Science and Technology of China. He is also the Cheung Kong Scholar Distinguished Professor. His current research interests include blockchain, big data security, and AI security.



Yu Jiang received the B.S. degree in software engineering from Beijing University of Posts and Telecommunications, Beijing, China, in 2010, and the Ph.D. degree in computer science from Tsinghua University, Beijing, in 2015.

He was a Postdoctoral Researcher with the Department of Computer Science, University of Illinois at Urbana–Champaign, Champaign, IL, USA, in 2016, and is now an Associate Professor with Tsinghua University. His research interests include domain specific modeling, formal computation model, formal verification and their applications in embedded systems.