HSTCG: State-Aware Simulink Model Test Case Generation with Heuristic Strategy

Zhuo Su*, Zehong Yu*, Dongyan Wang†, Yixiao Yang ‡, Rui Wang‡, Wanli Chang§, Aiguo Cui¶ and Yu Jiang ✉*

*KLISS, BNRist, School of Software, Tsinghua University, Beijing 100084, China
†Information Technology Center, Renmin University of China, Beijing 100872, China
‡Information Engineering College, Capital Normal University, Beijing 100089, China
§Department of Computer Science, University of York, York YO10 5DD, United Kingdom
¶HUAWEI Technologies Co. LTD., Shanghai 200120, China

***Abstract*—Simulink has gained widespread recognition as a valuable tool for system design. It supports efficient modeling and synthesis of embedded controllers. Test cases can also be automatically generated to simulate and verify the correctness of the Simulink model. Nevertheless, as systems grow increasingly complex, particularly in terms of their internal states, this complexity poses new challenges for existing model testing methodologies. Traditional techniques such as constraint solving and random search encounter difficulties when attempting to explore the intricate logic embedded within these models.**

In this paper, we introduce `HSTCG`, a state-aware test case generation method for Simulink models with heuristic strategy. `HSTCG` solves only one iteration of the model each time to get the test input that can cover a target branch, then executes the model once to obtain and update the new model state based on the solved input dynamically. Then, it solves the remaining branches based on the new model state iteratively until all the coverage requirements are satisfied. To improve the efficiency of test case generation, we also designed a heuristic strategy containing heuristic branch searching, repeated state filter and unreached branch filter to minimize the times of constraint solving. We implemented `HSTCG` and evaluated it on several benchmark Simulink models. Compared to the built-in Simulink Design Verifier and state-of-the-art academic work SimCoTest, `HSTCG` achieves an average improvement of 55% and 103% on Decision Coverage, 53% and 62% on Condition Coverage and 192% and 201% on Modified Condition Decision Coverage, respectively. We also validated the significant improvement of the heuristic strategy, which can improve the efficiency of test case generation by 62.2% on average.

***Index Terms*—Test case generation, Simulink, Constraint solving, Heuristic strategy**

I. Introduction

Simulink [2] stands as one of the foremost tools in the realm of model-driven design. Its widespread use extends increasingly into embedded scenarios [3]–[5]. Simulink has capabilities for efficient modeling, rapid simulation, and the generation of high-quality code tailored for embedded control models [6], [7]. To ensure the security and stability of these models, comprehensive testing is imperative [8]. However, manually crafting test cases proves to be a labor-intensive endeavor, often falling short of providing exhaustive coverage of the model's intricate elements. Automatic test case generation can save significant efforts and cover a lot of logic that is difficult to detect manually [9].

Presently, a substantial body of work has been dedicated to the realm of test case generation for Simulink models, as evidenced by numerous studies [10]–[14]. Broadly speaking, these endeavors can be categorized into two primary approaches. The first approach is rooted in the constraint solving method, exemplified by the Simulink Design Verifier (SLDV), an integral tool of the Simulink toolkit [15]. In this method, the Simulink model is typically transformed into a specific formal representation, followed by the application of a formal solver to meticulously address the constraints pertaining to the various branch logic within the model. The ultimate objective here is to derive model inputs that impeccably satisfy all constraints. The second approach is grounded in the random search method, typified by tools like SimCoTest [16]. In this methodology, input data for the model is generated randomly, and the model is subsequently executed to capture feedback coverage information. This feedback data is then employed to further refine and optimize the test case generation process.

While the prior research endeavors highlighted earlier have undeniably made significant strides in the domain of Simulink model testing, a noteworthy challenge remains when it comes to generating high-coverage test cases for models with intricate internal states. These models often involve numerous control conditions that necessitate specific states to be triggered, posing difficulties for traditional constraint solving methods. The inherent complexity associated with solving for these intricate model states often results in extended computation times, rendering it arduous to identify viable solutions within a reasonable timeframe. As for the random search method, it is also difficult to generate test cases that can reach the specific model states, even harder for the state-dependent conditions.

Figure 1 shows a view of a control model with complex internal states, which is an AutoSAR CPU task dispatch model. This model mainly contains a task queue, and the tasks in this queue are dynamically maintained through four operations, that is Add, Delete, Modify and Check, respectively. Task deletion, modification and checking require finding the item from the task queue that matches the task ID and the task parameter. Therefore, it is essential for the corresponding task item to already exist in the task queue before executing these three operations. For constraint solving methods, it is very time-consuming to obtain an input test case like "add data first and then modify data" directly. This is because the entire model logic and all model states are performed one more

This paper is an extended version of a conference paper [1]. Yu Jiang is the corresponding author.

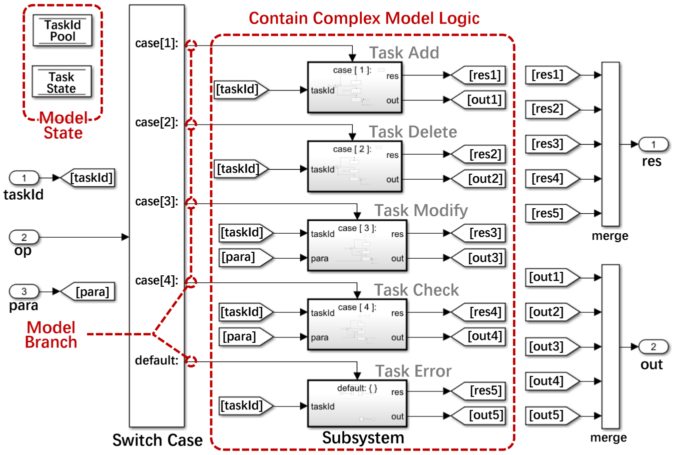


Fig. 1. An example model with complex internal states. This is an AutoSAR CPU task dispatch model. It mainly contains the four operations of adding, deleting, modifying and checking CPU task in the queue.

formal transformation for each additional solving step. These increased formal statements introduce exponential complexity to the solver [17]. For random search methods, since they usually involve random changes to the input data, there is only a small probability to generate a test input like “the previously added task ID matches the task ID to be modified later”.

To address the above problem, we propose a state-aware test case generation method with heuristic strategy. The key idea is to maintain a state tree to represent the execution paths, and solve the state-dependent condition based on the specific model state iteratively to avoid solving for the whole complex model states. First, it tries to solve one iteration of the model to obtain the input data that can trigger a target branch. Then, the input data is fed into the model for dynamic execution to obtain the new state of the model. The input data and the state are recorded as a new node in the state tree. After that, we continue to solve one iteration for the remaining branches based on the new state node. The loop will repeat until all coverage requirements are satisfied. In addition, a random trace is executed dynamically to explore a new state space when all the tree nodes are unable to be solved for a new coverage. Based on this state-aware test case generation method, the difficulty of constraint solving will be significantly reduced. Because we bring the model state as a constant into the solving process, the logic that depends on the model state will be explored easily. To improve the efficiency of test case generation by reducing the solving times, we also propose a heuristic strategy. This strategy involves three aspects, heuristic branch searching, repeated state filter and unreachable branch filter. The heuristic branch searching module tries to increase the solving priority of the branches associated with the new state. The repeated state filter module avoids duplicate solving by recording and comparing previous solve objectives. The unreachable branch filter module utilizes branch implication relations to avoid deep branch solving.

We implemented and evaluated HSTCG on several benchmark Simulink models. Compared to the built-in Simulink Design Verifier and the academic work SimCoTest, HSTCG achieves an average improvement of 55% and 103% on Decision Coverage, 53% and 62% on Condition Coverage and 192% and 201% on Modified Condition Decision Coverage(MCDC), respectively. Meanwhile, experiments show that the heuristic strategy can reduce the test case generation time by 62.2% on average.

II. Background and Related Work

A. Model-Driven Design

Model-driven design stands as a widely embraced software development methodology, especially within the realm of embedded systems. This approach comprises four fundamental components: behavior modeling, simulation, testing, and code generation [3], [18]–[22]. Behavior modeling encompasses the initial phase, wherein a formal model is constructed using either textual or graphical representations, tailored meticulously to meet the specific requirements of the user. Simulation plays a pivotal role in the design phase, serving as a means for debugging and verifying the functional correctness of the model, ensuring it aligns with the required functionality. Testing constitutes an integral step in the model-driven design process, providing a structured pathway for exhaustive testing and analyzing the model’s behavior and performance, thereby assuring its reliability. Code generation represents the critical stage where the model is converted into executable code, ready for deployment on embedded devices, thereby translating abstract concepts into practical, functional applications. In both academic and industrial contexts, a diverse array of design tools has emerged to bolster the model-driven design paradigm. Prominent examples include Ptolemy-II, Tsmart, and Polychrony in academic circles [23]–[25], as well as Simulink, SCADE, and DaVinci Developer in the industrial area [2], [26], [27]. Among these, Simulink, developed by MathWorks, is the most widely used tool, known for its robust capabilities in model simulation and code generation. Simulink also includes an extensive library of blocks carefully tailored to a wide range of industries, streamlining the system design and development process with remarkable efficiency.

B. Relevant Concepts of Simulink Model

To better understand the testing of Simulink models, we need to introduce some concepts about Simulink models, block, branch and internal state.

Simulink model is essentially a computation graph. As shown in Figure 1, it reads data from the inports on the left, performs a series of calculations, and outputs the results to the outports on the right. Data in the model is usually passed through the line, but sometimes, for aesthetic reasons, the “from goto” block pair is also used, such as the “taskId” and “res1” in the figure.

Block denotes a computation or control unit in a Simulink model, e.g., an Add block is a computation unit, and the SwitchCase block in the model of Figure 1 is a control unit. Alternatively, a composite subsystem that packages multiple blocks together can also be called a block. e.g., the five Subsystems in Figure 1. The Stateflow Chart, which allows state machine modeling, also belongs to the computation block. The “TaskId Pool” and “Task State” in the figure are special blocks, which are of data memory type and are used to define the global variables.

Branch represents all conditions in the model that can trigger different execution logic. It consists of two main types: (1) Branches in control blocks, which directly affect the scheduling of model blocks at the model level, can cause some blocks to execute or not. For example, the SwitchCase block can decide which Case subsystem to execute by its control port. Similarly the IfElse block. (2) Branches in computation blocks with internal decision logic, which typically make judgments within the block about the block’s input data or their own state values to influence the output. For example, a Saturation block limits the data based on a threshold value, and a Switch block selects one of two data inports for output based on the value of the control port. The logical blocks AND and OR also carry internal branches, and their branch condition is a judgment on the truth table. The state transitions in the Stateflow Chart also belong to this type of branch. The concept of a parent branch is embodied in the control flow of the model. A branch that can directly influence whether or not subsequent branches are executed is a parent branch. For example, assuming that the Task Delete subsystem in Figure 1 has an internal If block that determines whether or not a taskId exists, then the branch case[2] of the SwitchCase block in Figure 1 is the parent of the true and false branches of that If block. It is similar to branch nesting in code.

Internal state is variable data that is continuously retained by the model as it runs. Many Simulink blocks have internal states, such as the Delay block is used to delay data to be output at subsequent model iterations, the Ramp block outputs incremental data through its internal counter, the Accumulator block is used to accumulate data, and the current state of the Stateflow Chart is also an internal state. For the model as a whole, its internal state is the set of internal states of all the blocks it contains. Since the state may affect the execution of branches, this paper focuses on constraint solving for each branch based on the internal state of the model. For example, a Switch block makes a judgment on the output of an Accumulator block, and a Stateflow Chart transition makes a judgment on the current state.

C. Model Test Case Generation

Before introducing test case generation, we should be clear about how the model is executed and what the test cases look like. First, the model is executed iteratively. As in the CPUTask model in Figure 1, it will execute the logic from the inports on the left side to the outports on the right side over and over again. Each execution we call an iteration or a step. As for the test case, it is a sequence of data from the inports that will be required for the execution of each iteration. Each iteration takes the data from the sequence in turn to execute the model. For example, a sequence like $\{\{taskId_1, op_1, para_1\}, \{taskId_2, op_2, para_2\}\}$ allows the CPUTask model to execute two iterations.

Its significance is underscored by its role in ensuring the quality and dependability of both the model and the ensuing generated code [28]–[30]. The primary goal of testers often revolves around achieving comprehensive test coverage for the model [31]–[34]. In this pursuit, automated tools are commonly enlisted during the model testing phase, a strategic move that not only conserves manual effort but also enhances efficiency. The deployment of these tools facilitates the generation of an array of diverse test cases, enabling testers to find potential errors, bugs, and elusive corner cases lurking within the model. Within the domain of test case generation for model, two prominent methods emerge as significant players: constraint solving based test case generation and random search based test case generation.

(1) Constraint solving based test case generation. It usually uses formal techniques to obtain input cases that satisfy the property requirements. An example of such a technique is Simulink Design Verifier (SLDV) [15], an integrated validation toolkit within Simulink. SLDV leverages symbolic execution to automatically generate test cases aimed at fulfilling various model coverage criteria, such as Decision Coverage, Condition Coverage, Modified Condition Decision Coverage, and the derivation of custom test objectives. In the work presented by He et al. [10], a model checking approach is adopted to meticulously explore the structure of the target model. This exploration aims to identify a subset of nodes that maximizes the observation of mutated model blocks, subsequently leading to the generation of a concise set of test cases designed to attain high coverage based on this information. AutoMOTGen [11] takes a distinctive approach by describing the Simulink model using a formal language known as SAL [35]. It encodes coverage specifications within the formal model and utilizes built-in model checking tools to facilitate test case generation.

One common limitation encountered in these approaches is their inability to account for internal states when deriving test cases, particularly when internal states are employed as conditions. In contrast to these methods, HSTCG takes a novel approach. It records internal states obtained through dynamic execution and systematically solves state-aware branch conditions iteratively. This approach enables HSTCG to explore a more extensive state space and achieve higher coverage levels at an accelerated pace.

(2) Random search based test case generation. Random search methods are widely applied in the testing of large models [12]–[14], [16]. Typically, this approach relies on dynamic simulation to collect valuable test feedback. Reactis [13], for instance, employs Monte Carlo methods to generate test cases for random simulation. It also incorporates the guided simulation technique to evaluate output values, aiding in the selection of test cases for exploring uncovered blocks within the model. Another tool, REDIRECT [12], focuses on analyzing the feedback derived from the simulation of generated test cases. It employs a set of heuristics specifically tailored for non-linear blocks, enhancing its test case generation capabilities. In contrast, SimCoTest [16] stands out for its ability to generate test cases suitable for both continuous-time and discrete-time Simulink models.

However, when dealing with complex blocks and internal states nested within models, random search approaches often struggle to generate test cases capable of activating this deep

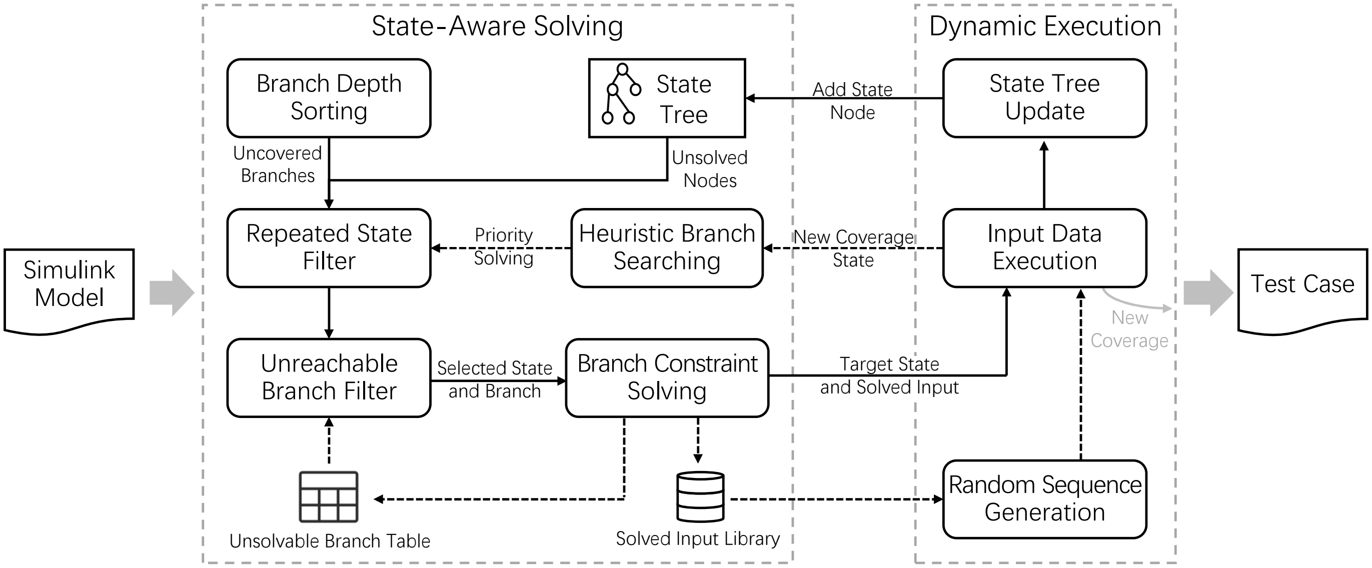


Fig. 2. Overview of HSTCG. Two main parts are executed cyclically to obtain test cases. The state-aware solving part focuses on obtaining the one-step input data by constraint solving on one iteration of the model. The dynamic execution part focuses on obtaining the specific state of the model by executing the solved input data and outputting test cases. The three modules, which are heuristic branch searching, repeated state filter and unreachable branch filter, aim to speed up the test case generation process by reducing the solving times.

logic and are unable to satisfy high-standard coverage criteria like MCDC [12], [14]. Different from them, HSTCG uses the constraint solving method with internal states to derive precise requirements for satisfying coverage criteria and generate corresponding test cases.

Difference from test case generation for regular software. Testing tools in the software domain can usually be classified into three types, dynamic search based methods, symbolic execution (constraint solving) based methods, and concolic testing that combines dynamic search and symbolic execution [36]. They usually target code and aim at coverage of code blocks (or branches). In contrast, testing of models is more demanding and considers more coverage metrics, including Decision Coverage, Conditional Coverage, and Modified Condition/Decision Coverage (MCDC). In addition, the execution logic of models is different from that of conventional software. Models usually need to be executed iteratively, which is equivalent to repeating the model logic in a while loop. The input data required by the model is field repetitive, which corresponds to the model inports. All these differences make it difficult for testing tools in the software domain to work on models. In other words, a lot of adaptation work is needed to be applied to model testing, such as code generation for models, test driver generation, model coverage statistics, model-oriented input data mutation, etc.

III. HSTCG Design

Figure 2 shows an overview of HSTCG. HSTCG takes the Simulink model as input and generates high-coverage test cases as output. The two main parts of the framework are executed iteratively to generate test cases. **Part one is State-Aware Solving**, which is mainly used to obtain the one-step input by state-aware branch constraint solving. It first initializes a state tree containing only one root node which represents the default state of the model and sorts the model branches by their depth. Then, it traverses the model branches and state tree nodes to perform state-aware solving. If input data is obtained to take a target branch from a state of interest, it will be taken to part two for execution. At the same time, the solved input will be stored in a library. **Part two is Dynamic Execution**, which is mainly used to obtain and update the model's internal state and synthesize test cases. The solved input data from part one will be brought into the target model state for execution. In case there is no result from part one, a random input sequence will be generated from the solved input library for multi-step execution on a random state within the state tree. After execution, the new model states will be added as a child node to the target state node. Once a new model branch is covered during the execution, all the input data on the current state tree path will be synthesized as a test case. The iteration of the state-aware solving and dynamic execution will continue until all the coverage requirements of the model branch are satisfied. **Three heuristic strategy modules** in the state-aware solving part are aimed to reduce unnecessary invocation of the constraint solver. The heuristic branch searching module searches for some branches related to new states and adds them to the priority solving queue. The repeated state filter module avoids repeated solving by determining whether the current state and branch have already been solved, i.e., whether the branch has been previously solved on that state. The unreachable branch filter module utilizes branch nesting relations to avoid solving branches inside unreachable branches. The next subsections will introduce the state-aware solving, the dynamic execution and three heuristic strategy modules, respectively.

A. State-Aware Solving

Before we introduce the detailed steps of state-aware solving of branch constraint solving, some important concepts need to be clarified first.

Definition 1 (model branch): The model branch B is defined as a tuple $\langle C, F, D\rangle$ which represents a decision for a block of the Simulink model that has condition logic. Among them, C is the condition to enter this branch, F is the parent branch of this branch, and D represents the branch depth which is the number of its all predecessor branches. For example, a Switch block in Simulink contains two branches, one is the decision when the value of its control port is true, and the other is the decision when it is false. In essence, constraint solving for a model branch is to find a model input that satisfies the constraints of the model branch and all its ancestor branches. The detailed method of calculating branch depth: Consider the model as a directed data flow graph starting from the top-level inport, find all the paths in the model of the block to which the branch belongs, and the branch depth is the number of branches contained in the block that contains the most branches among all the paths.

Definition 2 (model state): The model state S is defined as a tuple $\langle G, GV, M, ML, I, IV\rangle$ which represents the precise state of the model after each iteration. Among them, G and GV are the global variables and their values, respectively. M and ML are the state machines and their current locations, respectively. I and IV are the internal states of all actors and their state values, respectively. For example, the storage data of the Delay block and the last output value of the Ramp block in Simulink are recorded as model states. In our state-aware method, the model state values will be fixed as constants before each constraint solving.

Definition 3 (state tree node): A state tree node N is defined as a tuple $\langle P, S, IN, SB, CV\rangle$ which represents one of the possible states of the model. Among them, P is the parent node, S is the model state, IN is the input data that can cause the model to switch to state S based on the parent state P. SB is a set of the model branch, it records all model branches that have been solved in this state. CV represents the model branches covered by this state and all ancestor states confirmed by dynamic execution.

Definition 4 (state tree): The state tree T is a structure of a tree consisting of a set of state tree nodes $\{N_0, N_1, N_2, ..., N_n\}$. It represents all the model states that have been explored by `HSTCG`. The state tree contains a root node N_0 by default. This root node represents the initial state of the model, while additional nodes are dynamically added during the execution process. Each path within the tree serves as a real execution trace for the model, also serving as a distinct test case.

When the state-aware solving is first performed, the state tree needs to be initialized with a root node that contains the default state of the model. Meanwhile, the model branches need to be sorted by depth to accelerate the test case generation process. We prefer to perform constraint solving on shallow model branches. It is usually easier to perform solving for the shallow branches because the constraint solver will have to handle simpler logical expressions so that the solution can be obtained in less time. Moreover, the solved inputs that can cover shallow branches will sometimes cover deeper branches in the same execution. Therefore, it can avoid solving for those deeper branches, which can further reduce the time to reach a solution. More intuitively, e.g., for the CPUTask model in Figure 1, where the five branches of the SwitchCase block are the shallowest, solving them only requires making “op” equal to the corresponding case value. The branches in the subsystems, on the other hand, need to go through a more constraint solving process. For example, it takes 0.4 seconds to solve the “op” (that is, case[1]) corresponding to the Add Task operation of the Switch block of the top-level model. Solving for “Successfully Add” a CPU task inside the AddTask subsystem takes 0.5 seconds. But in fact, the result of both solving is the same. Then, we traverse all model states in the state tree and all branches of the model to perform state-aware constraint solving.

Algorithm 1: State-aware solving

```
Input: $Model$: The Simulink model for test case generation
    $BranchList$: The model branches after sorting by depth
    $StateTree$: The state tree during test case generation
Output: $TargetS$: The selected state node in $StateTree$
    $TargetB$: The selected branch in $BranchList$
    $SolvedInput$: The solved input of target state and branch
1  $SolvedInput$ = $NULL$
2  for $branch$ in $BranchList$ do
3      if isBranchCovered($branch$) then
4          continue
5      for $node$ in $StateTree$ do
6          if $node$.isSolved($branch$) then
7              continue
8          $Model$.setState($node$.getState()) // Switch model state
9          $SolvedInput$ = solve($Model$, $branch$) // Constraint
             solving for a branch on the current model state
10         $node$.setSolved($branch$)
11         if $SolvedInput$ != $NULL$ then
12             // The current state is solvable for target branch
13             $TargetS$ = $node$
14             $TargetB$ = $branch$
15             return $TargetS$, $TargetB$, $SolvedInput$
16 return $NULL$
```

The detailed state-aware solving process is shown in Algorithm 1. For the model branches, we only focus on those that have not been covered yet. The branches that have been covered by dynamic execution do not need to be solved, in lines 2-4. For the states in the state tree, we also avoid duplicate solving by determining whether the branch has already been solved on that state or not, in lines 5-7. Then, we try to solve for the state nodes and model branches that satisfy the above requirements. In line 8, the model state value will be taken from the state node, and the model state needs to be switched. We just bring the model state value as constants rather than variables into the model for solving. In line 9, we use the constraint solver to solve for the current branch of the model on the current state to obtain a one-step input. Then, the current branch is marked as solved on the current state node. If there is a solution, the current state node, the current branch, and the solved input will be output directly, lines 11-15. As no solution is obtained, it may be due to a timeout for solving or a solver failure. For this reason, we continue to traverse the state tree nodes and model branches to find a valid solution. If no state in the state tree can be solved for new branch coverage, the algorithm will return NULL. Then, the dynamic execution part will use the previously solved inputs to construct a random sequence to expand the state space.

We use Figure 3 as an example to explain the process of state-aware solving. We abstract the partial CPUTask model, i.e., the model state is an array, and the model branches include only the success add task and the success delete task. Assuming that the Add(Success) branch is currently solved and the corresponding state S1 is obtained, our goal is to solve the Delete(Success) branch based on S1. Next, is the specific state-aware solving process.

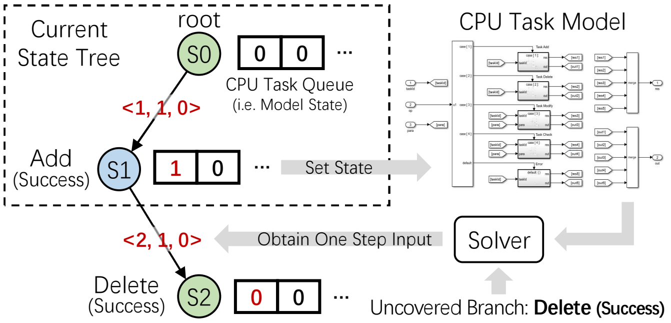


Fig. 3. An example of state-aware solving, corresponds to the CPU Task model in Figure 1. Assuming that the current state tree has two nodes, the initial state node and the Add(Success) node, we are now trying to solve the Delete(Success) branch. The squares to the right of each state node represent the abstract model state, i.e., the CPU task queue. The link between nodes represents one-step input. A new state can be obtained by executing one-step input on top of the source state.

First, for the Delete(Success) branch, we determine whether it has been solved on state S0. If it hasn't, we attempt to solve for one-step input based on the state S0 which can trigger it. Since no task exists in the CPU task queue on the state S0, the solving will obviously fail. We will not reflect the operation of solving failure in Figure 3. Next, we try to solve the Delete(Success) branch on the state S1. At this point, the state S1 already contains a CPU task with ID 1. We set the S1 state into the model. After that together with the model state, we transform the model into SMT formulas to solve the Delete(Success) branch. Since there is a task in the CPU task queue, we successfully solve it and get one-step input i.e. <op: 2, taskId: 1, para: 0>. In the later dynamic execution step, executing this one-step input can trigger the Delete(Success) branch and obtain the state S2. Task 1 in the CPU task queue is also deleted.

B. Dynamic Execution

The dynamic execution part can perform one-step execution for the solved input data of state-aware solving or multi-step execution using a random sequence of existing solved inputs in the library in case of no solutions. Once a set of input data triggers a new branch, it outputs all the input data on the current state tree path as a test case. After execution, it attaches new model states as child nodes to update the state tree for further iteration.

Algorithm 2: Dynamic execution

```
Input: $Model$: The Simulink model for test case generation
       $StateTree$: The state tree during test case generation
       $TargetState$: The selected state node of Algorithm 1
       $SolvedInput$: The solved input of Algorithm 1
       $InputLib$: All solved inputs from the solver
Output:$TestCase$:The test case that result in new coverage
1  $TestCase$ = ∅
2  $inputSequence$ = ∅ // Used for dynamic execution
3  $newCover$ = false
4  if $SolvedInput$ != $NULL$ then
5      // When state-aware solving has a solution
6      $curState$ = $TargetState$
7      $inputSequence$.append($SolvedInput$)
8  else
9      // When state-aware solving has no solution
10     $curState$ = $StateTree$.getRandomNode()
11     for $N$ times do
12         $inputSequence$.append($InputLib$.getRandInput())
13 for $input$ in $inputSequence$ do
14     $Model$.setState($curState$.getState()) // Switch model state
15     $newState$ = $Model$.run($input$)
16     $newCover$ = $newCover$ || hasNewCover($newState$)
17     $curState$.addChild($newState$) // Update state tree
18     $curState$ = $newState$
19 if $newCover$ then
20     // Get the complete input sequence
21     while $curState$ != $StateTree$.root do
22         $TestCase$.addData($curState$.getInput())
23         $curState$ = $curState$.getParentNode()
24 return $TestCase$
```

The detailed dynamic execution process is described in Algorithm 2. If the $SolvedInput$ from Algorithm 1 is not $NULL$, it will be used for one-step execution along with the selected state ($TargetState$). As shown in lines 4-7, this single input will be regarded as an input sequence. When state-aware solving has no solution, we construct a random sequence using the input data that has been previously solved, in lines 8-12. After that, the solved input from Algorithm 1 or randomly constructed input sequence will be brought into the model for dynamic execution. It is worth noting that when $SolvedInput$ is valid, we use the target state from Algorithm 1 as the current state for dynamic execution, in line 6. When $SolvedInput$ is invalid, we randomly select a state in the state tree, in line 10. The dynamic execution of the input sequence is shown in lines 13-18. It first switches the state data of the model to the current state. Then, it brings one input from the input sequence to execute the model once. A new model state and whether a new coverage can be triggered will be returned. The new state will be added as a child node of the current state so that the state tree is updated. The current state will be replaced with the new state to continue executing the input sequence. After execution, if a new coverage is found, the input data of the current node and all its parents will be output as a test case, as shown in lines 19-23.

C. Heuristic Strategy

1) Heuristic Branching Searching:

If only the exploration method of traversing all states and all unsolved branches as in Algorithm 1 is used, a lot of time may be wasted on meaningless solving. When we explore a new model state (which also means a new model branch is found), if we can learn which uncovered branches may be triggered based on this new state, we can prioritize solving it. However, for those branches that are not covered, it is difficult to directly know which of them can be triggered by a single step of execution based on that new state. So we can only use some potential indirect relationships to find those branches that are related to the new state. For this purpose, we design a heuristic branch search algorithm. With this heuristic method, those potentially related branches will be solved in advance, thus avoiding a lot of meaningless solving attempts.

As shown in Figure 2, when we obtain a new coverage state through dynamic execution, we search for uncovered branches of the model that may be related to this state to join the priority solving queue. Note that, the priority solving queue is not a concept from the field of computer science. It is just a first-in-first-out queue. After that, the state-aware solving step goes to solve the branches in the queue in priority based on the new state. To obtain the branches that are potentially related, we record those internal state variables that have changed since the last model execution and then traverse all branches of the model to check which branch decisions depend on those changed states. For example, in the CPU task dispatch model in Figure 1, when it successfully adds a task, the task modification needs to be performed based on the new state. What is reflected in the model is that the model rewrites an array of global variables in one iteration, and then this array of global variables has to be read and judged by another branch in the next iteration.

Algorithm 3: Heuristic branch searching

```
   Input: $Model$: The Simulink model for test case generation
          $BranchList$: The model branches after sorting by depth
          $NewState$: The new state that covers new branches
   Output: $PSQueue$:The queue for priority solving
 1 $PSQueue$ = ∅ // Each element is a key/value pair
 2 $parentState$ = $NewState$.parent
 3 $dataflow$ = $Model$.getDataflow()
     // Is used to find the data sources of branch blocks
 4 for $v$ in $Model.StateVarList$ do
 5     // Find model states (global variables) that have changed
 6     if $NewState$.value($v$) == $parentState$.value($v$) then
 7         continue
 8     for $branch$ in $BranchList$ do
 9         if isBranchCovered($branch$) then
10             continue
11         $dependentVarList$ = $dataflow$.getSrc($branch$)
             // Get variables that the branch condition depends on
12         if $dependentVarList$.find($v$) == $NULL$ then
13             // Only focus on model state (global variable)
14             continue
15         $PSQueue$.add($NewState$, $branch$)
16 return $PSQueue$
```

Algorithm 3 shows the specific process of heuristic branch searching. First, we define an empty priority solving queue, each element of which is in the form of “key-value”. “key” is the model state and “value” is the branch, indicating that the branch will be solved based on that model state in priority. After that, all state variables (global variables) of the model are traversed and the state values in the new state and the parent state are compared to find out which state variables have changed, as shown in lines 4-7. If we find a changed state variable, then we should add those uncovered branches whose condition depends on that state variable to the priority solving queue, as shown in lines 8-15. In detail, as shown in lines 11-14, we find out all the data sources of the block with branching judgment through the model’s dataflow and then determine whether the state variable being changed is in it. Finally, we pack the new state and the branch related to it into a key-value pair and add it to the priority solving queue without duplication, as shown in line 15.

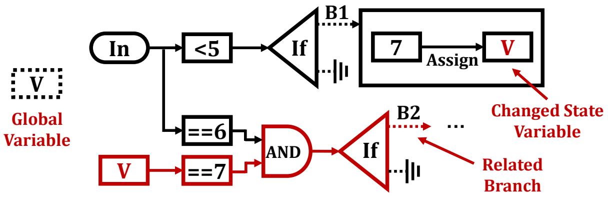


Fig. 4. Example of heuristic branch searching. V is a global variable. B1 is the branch that is currently solved. B2 is the branch that is affected by the variable V. B2 will be added to the priority solving queue.

Figure 4 shows an example demonstrating heuristic branch searching. This example contains a global variable V. Suppose we have solved B1 so far. Based on the results of the dynamic execution, we can learn that the variable V has been changed. Then we need to find those branches related to the variable V for priority solving. As shown in the block marked in red in the figure, B2 is the branch related to the variable V. It requires at least the value of V to be 7. Once B2 is added to the priority solving queue, it will be easily covered because V is assigned to be 7. Conversely, it will be impossible to solve B2 on states where the value of V has not been changed.

2) *Repeated State Filter:*

Throughout the HSTCG constraint solving process, the same branch may be solved on the same model state. This is because sometimes new branches are triggered that do not affect the internal state of the model, creating a new node with duplicate state in the state tree. For example, if the error branch of the CPU task model in Figure 1 is triggered, a new state node will be created. Its internal state is the same as that of the parent node because the error handling process only returns an error code. We cannot give up on creating these nodes with duplicate states, because each state node represents one execution step of the model. Therefore, we need to find out which solving can be skipped.

Figure 5 shows the details of the repeated state filter module. This module receives a target state node and a target branch ID that will be solved. They may come from the priority solving queue mentioned in the previous subsection, or obtained from the state tree. The Hash value of the model state (all variables) stored in the state node is computed first. We consider the set of those variable data as a byte array to perform the Hash operation. In this way, state nodes with the same Hash value can be considered to have the same state. After that,

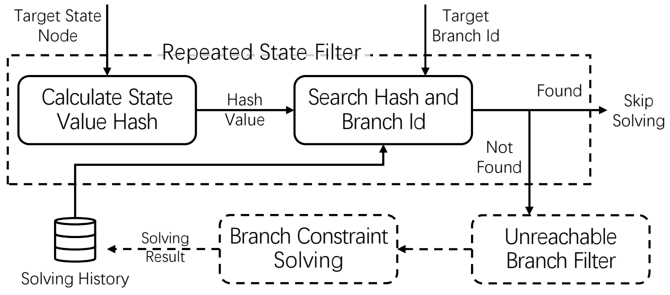


Fig. 5. Repeated State Filter. It avoids repeated solving by determining whether the target state and target branch have been solved.

we search for the pair of hash value and branch ID in the solving history. If found, we skip one solving by directly outputting the result saved in the solving history. If not found, we pass the target state node (Hash) and target branch ID to the subsequent modules which are unreachable branch filters and branch constraint solving. It is worth noting that this repeated state filter module enables successive test case generation functionality of `HSTCG`. Since the solving histories are stored in the form of files, these solving results can be used directly if test case generation is performed on the same model again.

3) Unreachable Branch Filter:

To further optimize the solving step in the test case generation process, we also exploit the implication relationship between the parent and child branches of the model to avoid solving unreachable branches. For example, when we fail to solve a branch on a specific state, then solving all the sub-branches of this branch on that state will fail. Since the solving for these sub-branches is meaningless, we should skip them.

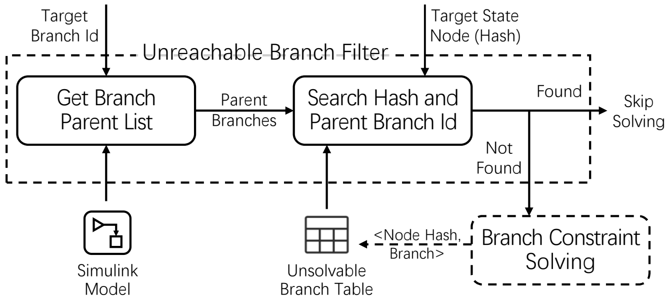
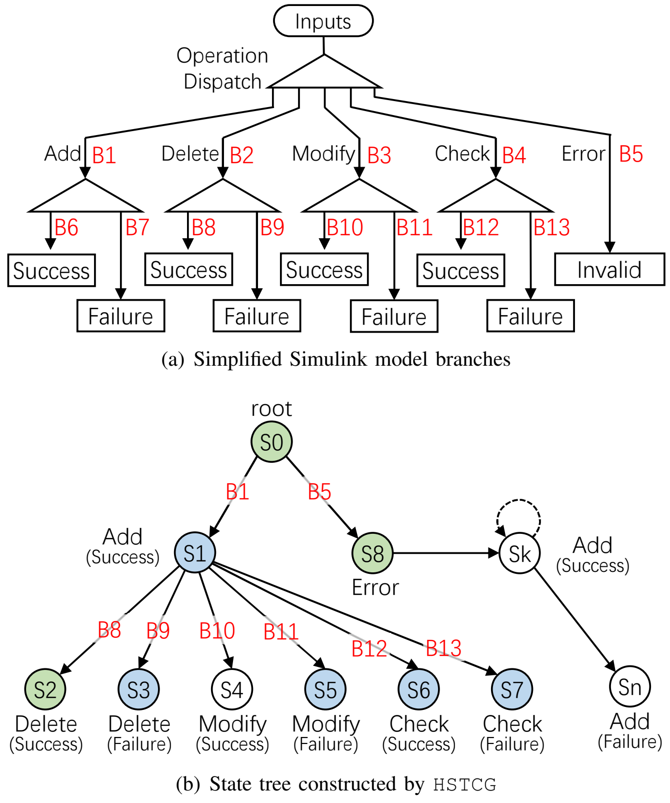


Fig. 6. Unreachable Branch Filter. It uses the implication relation of unreachable parent branch to avoid invalid solving.

Figure 6 shows the details of the unreachable branch filter module. It receives the state node (Hash) and branch ID passed from the previous modules. For a target branch, this module gets all its parent branches (ancestor branches) based on its structural relationship in the model. For example, the parents of the branch that determines whether the task queue is full in the “Task Add” subsystem in Figure 1 contains the Switch-Case branch at the outermost level of the model. After that, we search for each pair of target state Hash and parent branch ID from the unsolvable branch table. The pair to be searched for is like: $< N_1, B_1 >, < N_1, B_2 >, < N_1, B_3 >$, where N_1 denotes the state node hash value and B_x denotes the branch. Once a pair is found in the unsolvable branch table, then it is confirmed that the current target branch is unsolvable on the target state, and the solving of that branch can be skipped. Otherwise, it goes to the branch constraint solving module.

D. An Running Example of `HSTCG` Workflow

To illustrate the working process of `HSTCG` more clearly, we use the CPU Task model in Figure 1 as an example for analysis. Figure 7. (a) shows a simplified branch structure of the model. Since the model is based on opcodes to accomplish the corresponding functions, there are five branches at the first level, including the add, delete, modify, check operation of CPU tasks and the invalid operation. For each of the four operations, there are also two sub-branches, that is, operation success and operation failure. Note that the add operation will only fail when the CPU task queue is full, and the success of the delete, modify and check operation requires that there is a matched task in the queue. For this model, a possible state tree constructed by `HSTCG` is shown in Figure 7. (b). The corresponding test case generation process with heuristic strategy is shown in Table I.



(a) Simplified Simulink model branches

(b) State tree constructed by `HSTCG`

Fig. 7. An example of `HSTCG` corresponding to the model in Figure 1. (a) shows the simplified model branches, which contain a total of 13 branches. The triangle represents a judgement and each line below the triangle represents a branch. Among them, B1-B5 are shallow branches and B6-B13 are deep branches. (b) illustrates the state tree with the full coverage explored. S0-S8-Sk-Sn indicates the state nodes created in order. Nodes colored in green and blue in the state tree indicate that they have the same internal state ({S0, S2, S8}, {S1, S3, S5, S6, S7})

First, since the priority solving queue is empty, the B1 is solved on the root state S0. The execution condition of B1 is simple and only requires the opcodes of the input data to be

TABLE I
THE MAIN PROCESS OF TEST CASE GENERATION PROCESS WITH HEURISTIC STRATEGY

Stage	Target Branch	Target State	Achieved Branch	New State	Priority Solving Queue	Total Achieved Branch
1	B1	S0	B1, B6	S1	<S1: B7-B13>	`I....I.......`
2	Try to solve B7 on state S1, but failed.				<S1: B8-B13>	`I....I.......`
3	B8	S1	B2, B8	S2	<S1: B9-B13>, <S2: B7, B9-B13>	`II...I.I.....`
4	B9	S1	B9	S3	<S1: B10-B13>, <S2: B7, B10-B13>	`II...I.II....`
5	B10	S1	B3, B10	S4	<S1: B11-B13>, <S2: B7, B11-B13>, <S4: B7, B11-B13>	`III..I.III...`
6	B11	S1	B11	S5	<S1: B12-B13>, <S2: B7, B12-B13>, <S4: B7, B12-B13>	`III..I.IIII..`
7	B12	S1	B4, B12	S6	<S1: B13>, <S2: B7, B13>, <S4: B7, B13>	`IIII.I.IIIII.`
8	B13	S1	B13	S7	<S2: B7>, <S4: B7>	`IIII.I.IIIIII`
9	Try to solve B7 on state S2, but failed.				<S4: B7>	`IIII.I.IIIIII`
10	Try to solve B7 on state S4, but failed.				∅	`IIII.I.IIIIII`
11	B5	S0	B5	S8	∅	`IIIIII.IIIIII`
12	Skip solving B7 on S0, because the states of S0 and S2 are same.				∅	`IIIIII.IIIIII`
13	Skip solving B7 on S3, because the states of S3 and S1 are same.				∅	`IIIIII.IIIIII`
14	Skip solving B7 on S5, because the states of S5 and S1 are same.				∅	`IIIIII.IIIIII`
15	Skip solving B7 on S6, because the states of S6 and S1 are same.				∅	`IIIIII.IIIIII`
16	Skip solving B7 on S7, because the states of S7 and S1 are same.				∅	`IIIIII.IIIIII`
17	Skip solving B7 on S8, because the states of S8 and S0 are same.				∅	`IIIIII.IIIIII`
18	Failed to solve B7 on all state tree nodes.				∅	`IIIIII.IIIIII`
19	Random execution on S8.		B7	S9 - Sn	∅	`IIIIIIIIIIIII`

* The "`I`" in the last column represents the corresponding branch is covered, and the "`.`" represents uncovered. In the priority solving queue, a shorthand is used to express a series of elements, for example, "<S1: B10-B13>" should mean "<S1: B10>, <S1: B11>, <S1: B12>, <S1: B13>".

the target values. Therefore, based on the root state S0, it is easy to obtain the corresponding input data from the constraint solver. The deep branch B6 will also be covered following B1 due to dynamic execution. At this point, as a new state S1 is generated and the CPU task queue variable is changed, some related branches will be added to the priority solving queue. Specifically, except for the covered B6, all other branches of the success or failure operation, that is B7-B13, should be added to the priority solving queue. In contrast, B2-B5 are related only to the opcode and not to any model state, so they will not be added to the priority solving queue. Because a task exists in the CPU task queue in the S1 state, any add, delete, modify, or check operation on the task queue in the following stage is possible.

Then in stage 2, according to the priority solving queue, B7 is solved based on S1 in priority. However, we cannot get a valid solution from state S1 because the CPU task queue needs to be added more tasks to fill it up. In stage 3, according to the priority solving queue, B8 is solved based on S1 in priority. B2 will also be covered following B8 due to dynamic execution. Since a task in the CPU task queue was successfully deleted, some S2 state-related branches should be added to the priority solving queue. At the same time, S2 reverts to the same internal state as S0 since its task queue is also empty. In stage 4, since the B9 branch indicates a failure to delete a task, it does not affect the task queue, so there is no need to add any branch to the priority solving queue. Since the task queue is not modified, the internal states of S3 and S1 are the same. Similarly, B10-B13 can also obtain valid solutions on state S1.

In stages 9 and 10, according to the priority solving queue to solve B7 based on S2 and S4, but failed. In stage 11, since the priority solving queue is empty, we solve for B5 based on S0. Then in stages 12-17, we were originally supposed to try to solve B7 on the states that have not been solved for B7. However, since we have already solved on those nodes that have the same internal state, we just skip those solving that are certain to fail. When only B7 is left unsolvable on all state nodes, a random input sequence is constructed to execute dynamically using the previously solved inputs. Assuming that state S8 is chosen as the start state for random execution and the constructed sequence contains enough operations of adding CPU tasks, then we are able to cover B7 on the state Sn node eventually, as shown in stage 19. During the execution process of `HSTCG`, stages 1, 3-8, 11 and 19 will output the test cases.

To demonstrate the improvements brought by the heuristic strategy, we show in Table II the stages of the running example without the heuristic strategy. By comparing the two tables, we can find that the heuristic strategy reduces the number of solving times by 9 (from 20 to 11) than without the heuristic strategy. Among them, stage 12 (solve B8 on state S0), stage 15 (solve B10 on state S0) and stage 18 (solve B12 on state S0) in Table II is reduced by the heuristic branch searching module. The specific reason is that in the heuristic strategy, B8, B10 and B12 are added to the priority solving queue as soon as the S1 state appears. They can be solved directly based on the S1 state. The solving in stages 12-17 of the model Table I is reduced by repeated state filter module. Due to the model structure itself, the unreachable branch filter module is not used for this model.

TABLE II
THE MAIN PROCESS OF TEST CASE GENERATION PROCESS WITHOUT HEURISTIC STRATEGY

Stage	Target Branch	Target State	Achieved Branch	New State	Total Achieved Branch
1	B1	S0	B1, B6	S1	`I....I.......`
2	B2	S0	B2, B9	S2	`II...I..I....`
3	B3	S0	B3, B11	S3	`III..I..I.I..`
4	B4	S0	B4, B13	S4	`IIII.I..I.I.I`
5	B5	S0	B5	S5	`IIIIII..I.I.I`
6	Try to solve B7 on state S0, but failed.				`IIIIII..I.I.I`
7	Try to solve B7 on state S1, but failed.				`IIIIII..I.I.I`
8	Try to solve B7 on state S2, but failed.				`IIIIII..I.I.I`
9	Try to solve B7 on state S3, but failed.				`IIIIII..I.I.I`
10	Try to solve B7 on state S4, but failed.				`IIIIII..I.I.I`
11	Try to solve B7 on state S5, but failed.				`IIIIII..I.I.I`
12	Try to solve B8 on state S0, but failed.				`IIIIII..I.I.I`
13	B8	S1	B8	S6	`IIIIII.II.I.I`
14	Try to solve B7 on state S6, but failed.				`IIIIII.II.I.I`
15	Try to solve B10 on state S0, but failed.				`IIIIII.II.I.I`
16	B10	S1	B10	S7	`IIIIII.IIII.I`
17	Try to solve B7 on state S7, but failed.				`IIIIII.IIII.I`
18	Try to solve B12 on state S0, but failed.				`IIIIII.IIII.I`
19	B12	S1	B12	S8	`IIIIII.IIIIII`
20	Try to solve B7 on state S8, but failed.				`IIIIII.IIIIII`
21	Failed to solve B7 on all state tree nodes.				`IIIIII.IIIIII`
22	Random execution on S8.		B7	S9 - Sn	`IIIIIIIIIIIII`

* The "`I`" in last column represents the corresponding branch is covered, and the "`.`" represents uncovered.

IV. Evaluation

A. Tool Implementation

`HSTCG`[1] is implemented in C++, with 29,030 lines of code. We defined a C++ struct to represent the state tree node. In this struct, we use a "vector" structure to store the child nodes and dynamically allocated memory to store the state and input value of each iteration of the model. All state elements, such as global variables and state machine locations of the model, are stored in a "map" structure in the form of "key-value". The "key" represents the name of the state element (usually described using the full path of the element in the model). The "value" represents the attributes of the state element, such as data type, array length, etc. Only the values of the model states are stored in each state tree node, and they are stored in sequence to speed up the memory data access. When we need to switch the model state, we just need to read the state values in memory in order and set them to the corresponding elements of the model by mapping the "key-value" structure. Similarly, the inputs for each iteration of the model are stored in the state tree nodes by linear memory. As for constraint solving, we utilize the CBMC tool to convert the code of the Simulink model into SMT formulas, and then employ the MINISAT constraint solver to implement one-step-input solving.

The heuristic strategy is introduced before the branch constraint solving module in the state-aware solving step. The "state-branch" pairs in the priority solving queue are prioritized for solving before traversing all nodes in the state tree and all branches. In the implementation, the hash values used in the heuristic strategy are computed using the CRC64 algorithm [37] on the state variables at the time of state node creation. When dynamic execution is performed, the input data are sequentially parsed to the corresponding ports of the model. If a test case needs to be output for a state tree node, we can find a path to the root node directly through the node's parent pointer, and then merge all the input data stored at the nodes on the path and write it to a file. Test case files in text format can also be exported by `HSTCG`, so that a fair coverage comparison can be performed by using the Simulink test block named "Signal Builder" [38].

B. Experiment Setup

To evaluate the effectiveness of `HSTCG`, we conduct comparison experiments with the Simulink built-in validation toolkit SLDV and academic tool SimCoTest in terms of coverage results. Since other academic and commercial tools are not publicly available, we can not compare `HSTCG` with them. Besides, we conducted an in-depth investigation of the practical effects of our state-aware method. All experiments are performed on the same environment (Windows 10, Intel i7-8550U CPU, 16GB RAM, Simulink 2022b, Simulink Design Verifier 4.8) with the same duration (1 hour). Since SimCoTest and `HSTCG` include random strategies, we repeat the experiment 10 times to obtain the average coverage result for a fair comparison. It is worth noting that these tools end up achieving almost unchanged coverage over multiple repetitions of the experiment. This is because `HSTCG`'s random execution simply repeats the execution of existing data quickly, and there is no additional distribution of data to explore new branches. Whereas SimCoTest was able to cover the shallow model branches every time, the deeper branches had more stringent constraints that it never managed to cover. In terms of coverage efficiency, `HSTCG` has almost no difference between each repeated execution, as most of its time is spent on constraint solving, while random execution is very fast in comparison. SimCoTest's efficiency is quite different, but the overall trend of its coverage folds is basically the same. Since a single test case may result in a larger percentage increase in coverage, we have not plotted the spread of values in the folded line plot, given the image carrying capacity and aesthetics. All benchmark models are from Huawei's model library and they are deployed in embedded scenarios. Table III shows the detailed description of these models, including model functionality, number of branches, and number of blocks.

C. Evaluation on Coverage Rate

We used the most widely used Decision Coverage, Condition Coverage, and Modified Condition Decision Coverage (MCDC) to measure the effectiveness of test case generation for different tools [39], [40]. Decision coverage analyzes elements that represent decision points in a model, such as a Switch block or Stateflow states. It is concerned with whether different branches of a block with branching logic can be executed. Condition coverage analyzes blocks that output the

[1] The implementation and the benchmark models are uploaded on the anonymous website: https://anonymous.4open.science/r/STCG-9BB3.

TABLE III
THE DESCRIPTION OF BENCHMARK MODELS

Model	Functionality	#Branch	#Block
CPUTask	AutoSAR CPU task dispatch system	107	275
AFC	Engine air-fuel control system	35	125
TWC	Train wheel speed controller	80	214
NICProtocol	Vehicle NIC communication protocol	46	294
LANSwitch	LAN Switch controller	131	570
UTPC	Underwater thruster power control	92	214
LEDLC	LED matrix load control	94	270
TCP	TCP three-way handshake protocol	146	330
RAC	Robotic arm controller	179	667
SolarPV	Solar PV panel output control	55	131
EVCS	Electric vehicle charging system	72	163
FMTM	Factory Multi-point Temperature Monitor	89	152

logical combination of their inputs (for example, the Logical Operator block) and Stateflow transitions. A test case achieves full coverage when it causes each input to each instance of a logic block in the model and each condition on a transition to be true at least once during the simulation, and false at least once during the simulation. MCDC analyzes blocks that output the logical combination of their inputs and Stateflow transitions to determine the extent to which the test case tests the independence of logical block inputs and transition conditions. A test case achieves full coverage for a block when a change in one input, independent of any other inputs, causes a change in the block output. A test case achieves full coverage for a Stateflow transition when there is at least one time when a change in the condition triggers the transition for each condition. A higher coverage metric means a more comprehensive examination of the model.

Table IV shows the coverage of the test cases generated by the different tools for benchmark models. Compared to SLDV and SimCoTest, HSTCG improves the Decision Coverage for 12%-127% (avg. 55%) and 15%-513% (avg. 103%), Conditional Coverage for 15%-144% (avg. 53%) and 16%-144% (avg. 62%), and MCDC for 25%-900% (avg. 192%) and 51%-400% (avg. 201%), respectively. It can be seen that our method achieves good results on these control models with complex internal states. For example, HSTCG achieves 100% Decision Coverage and 100% Condition Coverage on the CPUTask and the UTPC model. As mentioned earlier, it is easy to obtain a solution like "add data first and then modify data" using a state-aware method, which is difficult to obtain by other methods. On several Simulink models, such as TWC, NICProtocol and LEDLC, HSTCG obtained coverage of 90%-98%, which is close to 100%. By dynamically debugging the models using test cases, we found that most of them were missed due to dead logic. For example, there is an unreachable branch in the model named LEDLC. This is mainly because there are only four LED states, and the Switch-Case block, which performs different control logic based on the LED states, has an additional default port that is never used beside the corresponding four ports.

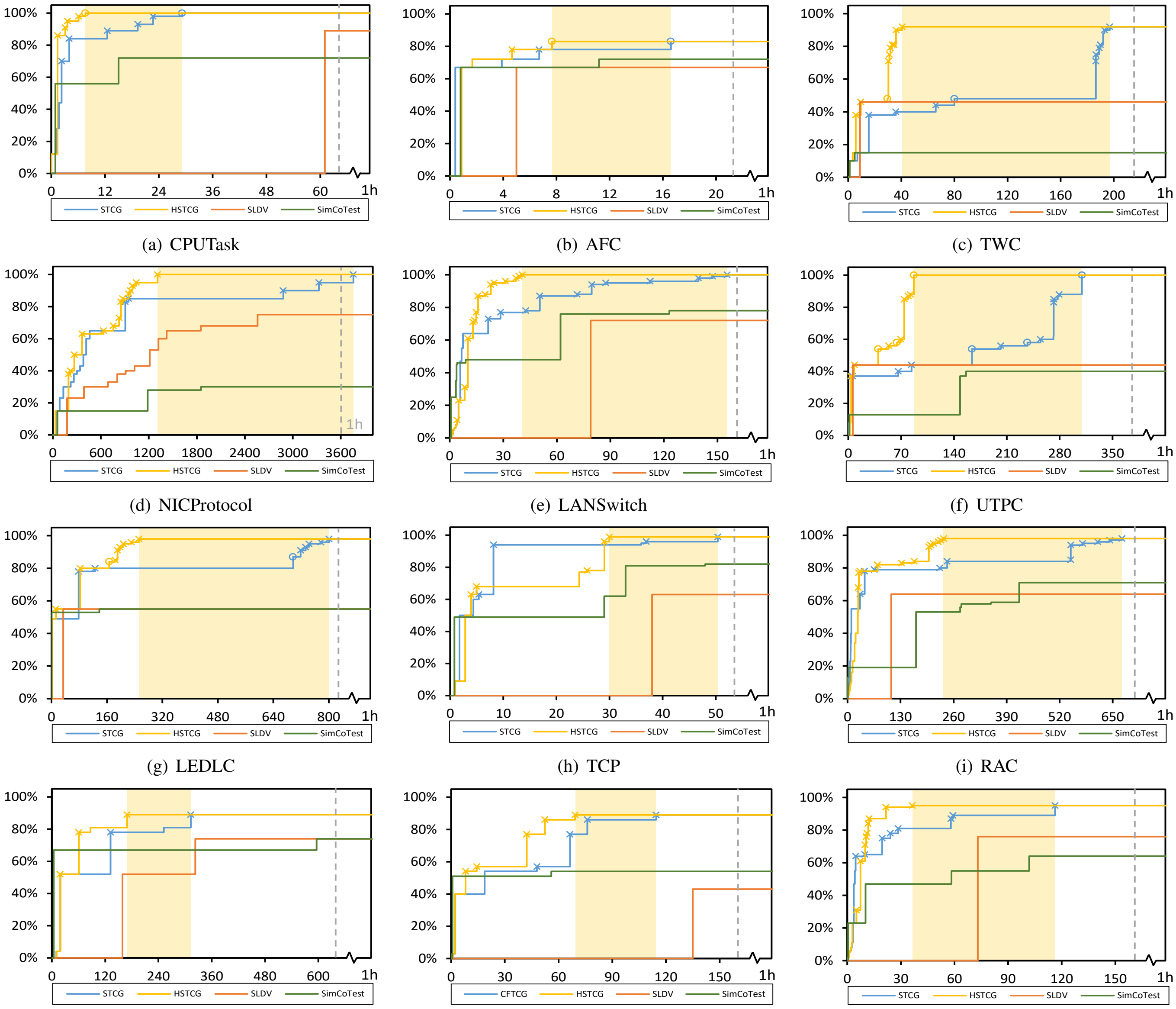
TABLE IV
COMPARISON OF THE TEST COVERAGE OF DIFFERENT TOOLS

Model	Tool	Decision Coverage	Condition Coverage	MCDC
CPUTask	SLDV	89%	72%	42%
	SimCoTest	72%	56%	21%
	HSTCG	**100%**	**100%**	**100%**
AFC	SLDV	67%	64%	11%
	SimCoTest	72%	68%	11%
	HSTCG	**83%**	**79%**	**22%**
TWC	SLDV	46%	68%	40%
	SimCoTest	15%	57%	20%
	HSTCG	**92%**	**97%**	**100%**
NICProtocol	SLDV	75%	83%	10%
	SimCoTest	30%	43%	33%
	HSTCG	**95%**	**98%**	**100%**
LANSwitch	SLDV	72%	76%	15%
	SimCoTest	78%	81%	15%
	HSTCG	**100%**	**98%**	**55%**
UTPC	SLDV	44%	59%	44%
	SimCoTest	40%	58%	44%
	HSTCG	**100%**	**100%**	**100%**
LEDLC	SLDV	55%	41%	43%
	SimCoTest	55%	41%	43%
	HSTCG	**98%**	**100%**	**100%**
TCP	SLDV	63%	64%	33%
	SimCoTest	82%	74%	17%
	HSTCG	**99%**	**100%**	**67%**
RAC	SLDV	64%	71%	12%
	SimCoTest	71%	76%	12%
	HSTCG	**98%**	**98%**	**23%**
SolarPV	SLDV	78%	83%	57%
	SimCoTest	74%	73%	47%
	HSTCG	**89%**	**95%**	**71%**
EVCS	SLDV	43%	37%	23%
	SimCoTest	54%	62%	23%
	HSTCG	**89%**	**87%**	**77%**
FMTM	SLDV	76%	77%	25%
	SimCoTest	64%	55%	15%
	HSTCG	**95%**	**95%**	**35%**
Average Improvement	vs SLDV	↑ 55%	↑ 53%	↑ 192%
	vs SimCoTest	↑ 103%	↑ 62%	↑ 201%

D. Test Case Generation Time

We also recorded the timestamp of each generated test case of three tools, HSTCG, SLDV and SimCoTest. Figure 8 shows the folded line of the Decision Coverage versus time for each Simulink model. To demonstrate the effectiveness of the state-aware method, we marked those test cases that were obtained by constraint solving based on internal model states (marked with "×") and those obtained from random sequence execution (marked with "○").

In Figure 8, we can see that in most cases, HSTCG is able to achieve higher coverage at a faster speed, and obtain new



(a) CPUTask

(b) AFC

(c) TWC

(d) NICProtocol

(e) LANSwitch

(f) UTPC

(g) LEDLC

(h) TCP

(i) RAC

(j) SolarPV

(k) EVCS

(l) FMTM

Fig. 8. The folded line plot of the Decision Coverage versus time. The X-axis is time (s) and the Y-axis is Decision Coverage (%). STCG is without heuristic strategy and HSTCG is with heuristic strategy. The yellowish areas in the background indicate the improvement impact of the heuristic strategy. "×" indicates test cases generated by constraint solving based on internal model states, and "○" indicates test cases generated by executing the random input sequence.

test cases continuously. In contrast, SLDV outputs test cases only once on most models. Although SLDV outputs test cases more consistently on the NICProtocol model, it is harder and harder to output new test cases due to the complexity of the deeper state of the model. Since SimCoTest does not require time-consuming constraint solving, it will obtain relatively high coverage at the beginning of the test, while it will be difficult to achieve coverage of state-dependent branches subsequently. More importantly, as seen from our mark of the test cases generated by `HSTCG`, the higher coverage fraction is almost always obtained by our state-aware branch solving. For example, on the TCP model, `HSTCG` can obtain the various handshake states of the client IP. Therefore, it is easy to solve the relevant branches of the second or the third handshake based on the existing handshake states.

E. Effectiveness of Heuristic Strategy

To better illustrate the effectiveness of the heuristic strategy, we also show the folded line plot of `STCG` (without heuristic strategy) in Figure 8. We use the yellowish areas in the background to indicate the improved effects of the heuristic strategy. We can see that all the experimental models benefit from the heuristic strategy. The time spent on test case generation for each experimental model is significantly reduced. The specific improvements in test case generation efficiency on each experimental model are shown in Table V.

According to Table V, we can see that the heuristic strategy reduces the test case generation time by 62.2% on average on these benchmark models. To explore whether these improvements were brought about by the heuristic strategy, we also recorded the number of solving of each model in the

TABLE V
HSTCG vs. STCG, The number of solving and solving times reduced by the three heuristic strategies

Model	Reduced # of Solving	Reduced # by	Reduced # by	Reduced # by	Reduced Time of Solving
	Rate (Before → After)	Repeated Solving	Invalid Solving	Heuristics Search	Rate (Before → After)
CPUTask	68.8% (44 → 15)	7	0	22	74.0% (29.2s → 7.6s)
AFC	72.7% (44 → 12)	0	0	7	53.6% (16.6s → 7.7s)
TWC	80.1% (357 → 68)	38	210	41	79.4% (197.1s → 40.7s)
NICProtocol	45.3% (95 → 52)	6	0	37	65.2% (3758.3s → 1308.4s)
UTPC	84.2% (646 → 102)	32	502	10	72.0% (309.6s → 86.8s)
LANSwitch	75.4% (175 → 43)	33	79	20	73.9% (155.4s → 40.6s)
LEDLC	75.2% (343 → 85)	8	244	6	68.5% (801.1s → 252.2s)
TCP	62.4% (101 → 38)	15	41	7	40.5% (50.4s → 30.0s)
RAC	80.7% (990 → 191)	174	559	66	65.0% (672.9s → 235.7s)
SolarPV	44.4% (36 → 20)	13	0	3	45.8% (312.8s → 169.4s)
EVCS	50.5% (111 → 55)	50	1	5	39.5% (114.5s → 69.3s)
FMTM	66.0% (197 → 67)	20	15	95	68.7% (116.3s → 36.4s)
Average	67.1%	-	-	-	62.2%

* Columns 3-5 denote the number of reductions in the number of solving by the three heuristics proposed in this paper, respectively.

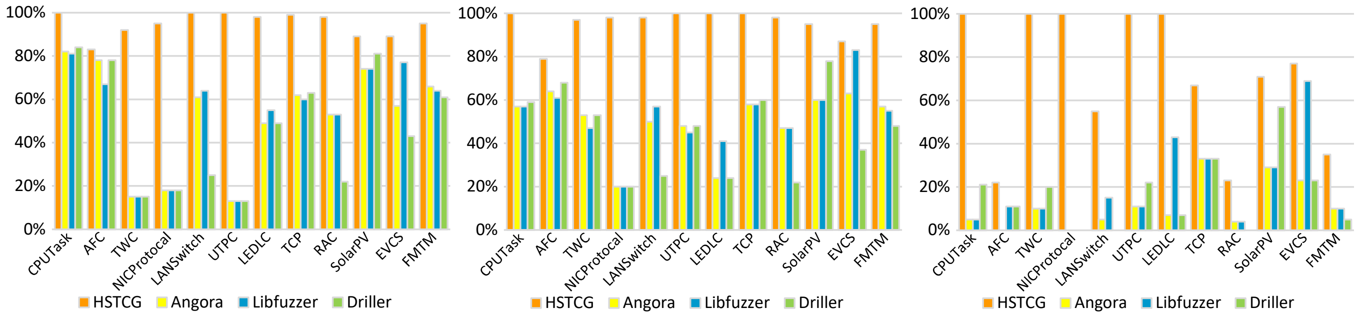
experiment, as shown in Table V. We can find a positive correlation between the time reduction in test case generation and the reduction in the number of solving. Further, we also recorded the number of repeated solving and the number of invalid solving that was reduced, corresponding to the repeated state filter module and the heuristic branch searching module, respectively. The reduced number of solving attempts due to heuristic search is also calculated by subtraction and is shown in the "Reduced # by Heuristics Search" column in Table V. We can see that each of these heuristic strategy modules plays an important role on different experimental models. Just as we envisioned in the design section, the heuristic strategy can reduce a large number of solving.

F. The Comparison with TCG methods in Software Domain

There are numerous works on test case generation, especially for software or code. So, we would like to know if it is possible to achieve better results using advanced testing works in the software domain to test the model. To address this question, three well-known testing tools Libfuzzer, Angora, and Driller were selected for additional comparative experiments [41]–[43]. To make them possible to generate test cases against the model, we utilized the code generated by the model for testing. We handwrote the corresponding test drivers and rewrote the interface to the model code so that the test tools can automatically provide test files for executing the code. After executing the test case generation process, we converted the binary test case files into Excel format to provide coverage statistics to Simulink. The results of the comparison with the test case generation works in software domain are shown in Figure 9. Compared to Libfuzzer, HSTCG was on average 39%, 40% and 47% higher in Decision Coverage, Condition Coverage, and MCDC, respectively. Compared to Angora, HSTCG was on average 42%, 45% and 59% higher, respectively. Compared to Driller, HSTCG was on average 48%, 50% and 54% higher, respectively.

This result shows that the testing methods in software domain are not suitable for testing Simulink models. We concluded that this is due to the following reasons. First, the test methods in software domain are not effective in generating test cases for the coverage metrics in the model. For example, the ABS block in Simulink has two decision coverage metrics, value$<$0 and value$\geq$0. The abs statement does not reflect any branch jumps, either in C code or assembly language. There are many similar cases, such as AND and OR operations, which result in a large number of missing coverage metrics for these works. Second, the random strategy of these works makes it difficult to guarantee the validity of the model inputs. Because the input data to the model is received through ports and the model is executed iteratively, the data inside the file we provide to the model-generated code is compactly arranged in the order of ports. Then, as soon as some bytes are inserted or removed from it, the data that follows will not correspond to the original ports anymore. This makes the random mutation approach of these works ineffective.

Based on the above analyzes, we think that a lot of additional work needs to be done to effectively apply software testing methods to models. We briefly tried the idea based on Libfuzzer. This involved manually instrumenting model coverage metrics that were missing from the code, and adapting Libfuzzer's mutation strategy to align to the input length of each iteration of the model. Preliminary experimental results show that this method achieves average coverage 15%, 14%, and 6% lower than HSTCG on Decision Coverage, Condition Coverage, and MCDC, respectively, on the benchmark models. The results demonstrate that it is useful for improving traditional software testing methods. As for those model metrics that are still not covered, they generally have higher



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

Fig. 9. Comparison of coverage with test case generation methods in the software domain. Libfuzzer is an in-process coverage-guided fuzzing tool. Angora is a fuzzing tool that integrates taint analysis technique. Driller is a concolic testing tool that combines fuzzing with symbolic execution.

requirements on the model state. To further improve coverage, perhaps it need to introduce constraint solving techniques to explore those complex logics.

Despite the lower coverage on the model of the software test methods, it is worth noting that most of the coverage obtained by these works is obtained at the moment the tool is launched, which is faster compared to HSTCG at the beginning. This shows that random algorithms can be very useful in the early stages of test case generation. For this reason, the random strategy will be integrated into our future work.

G. The Scalability on Larger Simulink Model

In real industrial scenarios, a complete model is usually a combination of multiple subsystems. For example, in the AutoSAR standard for vehicles, one SWC can contain several Runables, and developers often need to test the entire system in order to test whether the subsystems work in coordination with each other. The size of the complete model can be very large and can even contain thousands of model blocks. This poses a potential threat to our work. After all, HSTCG mainly uses formal constraint solving method. This is because the larger the solving object is the less friendly it is to constraint solving methods. To verify the capability of HSTCG on large Simulink models, we found three large models on the publicly available model set [44]. As shown in Table VI, TCS1 (containing 1743 blocks) and TCS2 (containing 7196 blocks) are control models dedicated to testing and analysis from the Simulink website. SVPWM (containing 1233 blocks) is a space vector pulse width modulation model commonly used for motor control.

The comparison results on the large Simulink model are displayed in Table VI. Compared to SLDV, HSTCG shows an average improvement of 32%, 35% and 135% in Decision Coverage, Conditional Coverage, and MCDC, respectively. Compared to SimCoTest, HSTCG shows an average improvement of 28%, 17% and 55%, respectively. It can be seen that HSTCG can still work well on large models. It is worth noting that HSTCG's average solving time on large models has increased. For the models in Table III, the average single solving time for HSTCG is 0.7 seconds, but the average single solving time on the three large models is 3.6 seconds. This illustrates that the large models indeed introduce more time overhead to HSTCG. However, the percentage increase in solving time is less than the model size. The average number of blocks for the models in Table III is 283, while the average number of blocks for the three large models is 3,390. HSTCG only increases the solving time by about 5x for a nearly 12x increase in model size. This is mainly because when we formally transformed the model, we removed as many blocks as possible that were not relevant to the current solving goal. This allows HSTCG to remain workable on large models.

TABLE VI
COMPARISON OF THE COVERAGE ON LARGER SIMULINK MODELS

Model	Tool	Decision Coverage	Condition Coverage	MCDC
TCS1 1,743 blocks	SLDV	60%	65%	16%
	SimCoTest	86%	68%	28%
	HSTCG	**99%**	**85%**	**41%**
TCS2 7,196 blocks	SLDV	79%	83%	43%
	SimCoTest	79%	87%	56%
	HSTCG	**85%**	**94%**	**92%**
SVPWM 1,233 blocks	SLDV	45%	55%	-
	SimCoTest	34%	75%	-
	HSTCG	**55%**	**89%**	-
Average Improvement	vs SLDV	↑ 32%	↑ 35%	↑ 135%
	vs SimCoTest	↑ 28%	↑ 17%	↑ 55%

* Since the SVPWM model does not contain condition combinations, it has no MCDC metric.

V. DISCUSSION

A. The limitation on Unreachable Branches

In our experiments, we found that some branches in the model, such as the TWC and LEDLC model, could not be triggered even after a long solving time and random execution. This problem exists in both HSTCG and STCG. After analyzing the models manually, we found that this situation is caused by the branch conditions being perpetually false (perpetually unreachable, unreachable in any state or dead logic). For example, if a Saturation block with lower and upper limits set to 0 and 5 is followed by another Saturation block with an upper limit of 6, then the branch of "value greater than 6" in the following Saturation block is an unreachable branch. Because these branches will be solved on each state in the state tree to determine if they are reachable on a known model state, HSTCG and STCG perform multiple solving for this type of branch, resulting in a lot of wasted time. However, this problem has been mitigated due to HSTCG's heuristic strategy.

This is because the solving of the same unreachable branch on the repeated state has been filtered and the sub-branches of the unreachable branch are skipped for solving. Nevertheless, to further address this problem, it is advisable to verify the unreachable branches using formal methods to get branches that are unsolvable on any state in advance. In this way, invalid solving can be further avoided.

B. The Selection of Constraint Solver

Since our tool employs a constraint solver and the constraint solving takes up most of the time in the tool run. So we also tried other solvers other than MINISAT to see if we could get faster results. We experimented with Z3 [45] and CVC4 [46] solvers respectively. We do not use solvers such as SPIN [47] and SMV [48] because we target concrete model logics and thus we would not need to abstract the model. The results showed that the average time per solving increased by 28% after replacing Z3. Z3 was faster compared to MINISAT only on 3 models. After replacing CVC4, the average time per solving increased by 19%. CVC4 was faster compared to MINISAT only on 2 models. According to our analysis, Z3 and CVC4 contain a lot of advanced features that are more suitable for more complex problems involving various theories and logic. Simulink is inherently limited by its modeling capabilities and does not contain data representations such as pointers or complex logic such as function calls and while-break loop. In contrast, MINISAT is a lightweight and efficient solver that is a better choice than Z3 and CVC4 for solving simpler problems.

C. The Possible Further Optimization

By comparing with SimCoTest, Libfuzzer, Angora and Driller, we found that the random search method is usually able to explore some coverage earlier than HSTCG, as shown in Figure 8. This is due to the random generation of test cases is faster than constraint solving. Meanwhile, since the random strategy of our algorithm is performed when all uncovered branches are unsolvable on all states, it results in the random strategy only working after a long time when testing the large Simulink models. If the random method can be introduced into HSTCG to perform the random generation process first and then use the constraint solver to solve the remaining uncovered branches, the efficiency of HSTCG can be further improved.

In addition, we found that there are cumulative logics in some models, such as statistics of cumulative errors and numerical cumulative overflow. Most of them require so many model iterations to trigger. With the state exploration approach of HSTCG, it is difficult to trigger those logics that require a long time to accumulate, even with a random sequence generation. For this reason, designing a long test case generation method for cumulative logic is also our future work.

VI. Differences from Preliminary Paper

This paper is an extended and revised version of a preliminary conference paper [1] (STCG). In terms of paper contributions, this paper adds a heuristic strategy to improve the efficiency of test case generation. This heuristic strategy contains three modules, heuristic branch searching, repeated state filter and unreachable branch filter. First, the heuristic branch searching module prioritizes branch solving based on the global variables that have changed in the new state and their associated branches. Second, the repeated state filter module avoids repeated solving by recognizing whether the state is to be solved and the target branch has been solved previously. Third, the unreachable branch filtering module avoids invalid solving by detecting whether an ancestor branch of the current target branch is unreachable in the current state. Not only that, we have added more experimental models to further validate the effectiveness of our tool. The experiments show that the state-aware test case generator with heuristic strategy achieves a significant efficiency improvement over previous work, reducing the number of constraint solving times by 67.1% and the test case generation time by 62.2% on average.

Heuristic Branch Searching: In comparison, STCG blindly traverses the state nodes on the state tree and all unsolved branches when performing stepwise constraint solving. In contrast, this paper proposes a heuristic branch searching module that effectively reduces the number of constraint solving times, thereby enhancing the efficiency of test case generation. When a new model state is triggered, we add the branches that are related to the new state at the model dataflow level to the priority solving queue. Specifically, to obtain the branches that are potentially related, we record those internal states of the model that have changed and then traverse all branches of the model to check which branch decisions depend on those changed states.

Repeated State Filter: In previous work (STCG), we did not ensure that the model states recorded in each state node on the state tree were different. That is, when a new branch is successfully explored, a state node is added to the state tree. However, if the triggering of this branch does not affect any global variables in the model, then two nodes with the same state will exist in the state tree. This makes it possible to solve on the same state multiple times when solving for a branch, which is obviously very time-consuming. For this reason, we designed the repeated state filter module to address this problem. It avoids time-consuming constraint solving by recording the solved state and branch pairs and comparing the model state recorded in the state node, and obtaining the previous solving results directly once the current target state and target branch have been solved.

Unreachable Branch Filter: In previous work (STCG), we may have solved for multiple branches on the same state, where some of the branches may not have been necessary due to the implication of unreachable branches. For example, if a branch is unreachable on the current state, then all the sub-branches contained in that branch must also be unreachable on the current state. For this reason, we design the unreachable branch filter module. It avoids invalid solving by recording those unreachable branches that are solved and comparing whether the branch to be solved is a sub-branch of those unreachable branches. In addition, since the branches are solved in order from shallow to deep, invalid solving can be avoided as much as possible.

VII. Conclusion

In this paper, HSTCG is proposed to optimize the test case generation of Simulink models with state-aware solving, especially for the control models that have complex internal states. More specifically, solving for only one iteration of a model state can simplify the solving difficulty and complexity. We can obtain model inputs for new coverage based on specific model states more easily. Dynamic execution using random input sequences in the absence of a solved result can further expand the exploration space. Three heuristic strategy modules are used to improve testing efficiency by reducing the number of solving in the test case generation process. Experiments show that HSTCG can perform well on benchmark Simulink models. Compared to SLDV and SimCoTest, the Decision Coverage from HSTCG can be improved by 55% and 103%, the Condition Coverage can be improved by 53% and 62%, and the MCDC can be improved by 192% and 201%, respectively. In addition, the heuristic strategy can avoid many times of solving, thus speeding up the test case generation process and reducing the time by about 62.2% on average. Our future work includes the use of formal techniques to avoid the solving of unreachable branches and the use of random strategies to reduce the number of solving times.

VIII. Acknowledgment

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

References

[1] Z. Su, Z. Yu, D. Wang, Y. Yang, R. Wang, W. Chang, A. Cui, and Y. Jiang, “STCG: state-aware test case generation for simulink models,” in *Design Automation Conference*, 2023, pp. 1–6.

[2] The MathWorks Inc., *Simulink Documentation*, accessed: 2023-10-17. [Online]. Available: https://www.mathworks.com/help

[3] F. Pasic, “Model-driven development of condition monitoring software,” in *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, 2018, pp. 162–167.

[4] D. K. Chaturvedi, *Modeling and simulation of systems using MATLAB and Simulink*. CRC press, 2017.

[5] S. Staroletov, N. Shilov, V. Zyubin, T. Liakh, A. Rozov, I. Konyukhov, I. Shilov, T. Baar, and H. Schulte, “Model-driven methods to design of reliable multiagent cyber-physical systems,” in *Proc. of the Conference on Modeling and Analysis of Complex Systems and Processes*, 2019.

[6] J. Krizan, L. Ertl, M. Bradac, M. Jasansky, and A. Andreev, “Automatic code generation from matlab/simulink for critical applications,” in *Canadian Conference on Electrical and Computer Engineering*, 2014, pp. 1–6.

[7] Z. Yu, Z. Su, Y. Yang, J. Liang, Y. Jiang, A. Cui, W. Chang, and R. Wang, “Mercury: Instruction pipeline aware code generation for simulink models,” *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 41, no. 11, pp. 4504–4515, 2022.

[8] F. Elberzhager, A. Rosbach, and T. Bauer, “Analysis and testing of matlab simulink models: a systematic mapping study,” in *International Workshop on Joining AcadeMiA and Industry Contributions to testing Automation*, 2013, pp. 29–34.

[9] A. Belinfante, L. Frantzen, and C. Schallhart, “14 tools for test case generation,” in *Model-based testing of reactive systems*. Springer, 2005, pp. 391–438.

[10] N. He, P. Rümmer, and D. Kroening, “Test-case generation for embedded simulink via formal concept analysis,” in *Design Automation Conference,*, 2011, pp. 224–229.

[11] S. Mohalik, A. A. Gadkari, A. Yeolekar, K. C. Shashidhar, and S. Ramesh, “Automatic test case generation from simulink/stateflow models using model checking,” *Softw. Test. Verification Reliab.*, vol. 24, no. 2, pp. 155–180, 2014.

[12] M. Satpathy, A. Yeolekar, and S. Ramesh, “Randomized directed testing (REDIRECT) for simulink/stateflow models,” in *ACM & IEEE International conference on Embedded software*, 2008, pp. 217–226.

[13] R. Cleaveland, S. A. Smolka, and S. Sims, “An instrumentation-based approach to controller model validation,” in *Model-Driven Development of Reliable Automotive Services*, ser. Lecture Notes in Computer Science, vol. 4922. Springer, 2006, pp. 84–97.

[14] A. Arrieta, S. Wang, U. Markiegi, G. Sagardui, and L. Etxeberria, “Search-based test case generation for cyber-physical systems,” in *Congress on Evolutionary Computation*, 2017, pp. 688–697.

[15] H. Grégoire, “Simulink design verifier-applying automated formal methods to simulink and stateflow,” in *Third Workshop on Automated Formal Methods*, 2008.

[16] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann, “Simcotest: a test suite generation tool for simulink/stateflow controllers,” in *International Conference on Software Engineering*, 2016, pp. 585–588.

[17] V. V. D’Silva, D. Kroening, and G. Weissenbacher, “A survey of automated techniques for formal software verification,” *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 27, no. 7, pp. 1165–1178, 2008.

[18] 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 *International Workshop on Software Engineering Research & Practices for the Internet of Things*, 2019, pp. 25–32.

[19] K. Jahed and J. Dingel, “Enabling model-driven software development tools for the internet of things,” in *International Workshop on Modelling in Software Engineerings*, 2019, pp. 93–99.

[20] F. Rademacher, J. Sorgalla, S. Sachweh, and A. Zündorf, “A model-driven workflow for distributed microservice development,” in *ACM/SIGAPP Symposium on Applied Computing*, 2019, pp. 1260–1262.

[21] Y. Jiang, H. Song, Y. Yang, H. Liu, M. Gu, Y. Guan, J. Sun, and L. Sha, “Dependable model-driven development of CPS: from stateflow simulation to verified implementation,” *ACM Trans. Cyber Phys. Syst.*, vol. 3, no. 1, pp. 12:1–12:31, 2019.

[22] Y. Jiang, H. Zhang, Z. Li, Y. Deng, X. Song, M. Gu, and J. Sun, “Design and optimization of multiclocked embedded systems using formal techniques,” *IEEE Trans. Ind. Electron.*, vol. 62, no. 2, pp. 1270–1278, 2015.

[23] 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*, ser. Systems on Silicon. San Francisco: Morgan Kaufmann, 2002, pp. 527–543.

[24] Y. Jiang, H. Zhang, H. Zhang, X. Zhao, H. Liu, C. Sun, X. Song, M. Gu, and J. Sun, “Tsmart-galsblock: a toolkit for modeling, validation, and synthesis of multi-clocked embedded systems,” in *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 711–714.

[25] P. Le Guernic, J. Talpin, and J. L. Lann, “POLYCHRONY for system design,” *J. Circuits Syst. Comput.*, vol. 12, no. 3, pp. 261–304, 2003.

[26] G. Berry, “Scade: Synchronous design and validation of embedded control software,” in *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*. Springer, 2007, pp. 19–33.

[27] Vector Informatik GmbH, *DaVinci Developer*, accessed: 2023-10-17. [Online]. Available: https://www.vector.com/us/en-us/products/solutions/autosar-classic/

[28] E. Bartocci, N. Manjunath, L. Mariani, C. Mateis, and D. Nickovic, “Cpsdebug: Automatic failure explanation in CPS models,” *Int. J. Softw. Tools Technol. Transf.*, vol. 23, no. 5, pp. 783–796, 2021.

[29] S. Anand, E. K. Burke, T. Y. Chen, J. A. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, “An orchestrated survey of methodologies for automated software test case generation,” *J. Syst. Softw.*, vol. 86, no. 8, pp. 1978–2001, 2013.

[30] J. Peleska, E. Vorobev, and F. Lapschies, “Automated test case generation with smt-solving and abstract interpretation,” in *NASA Formal Methods - Third International Symposium*, ser. Lecture Notes in Computer Science, vol. 6617, 2011, pp. 298–312.

[31] W. Li, F. Le Gall, and N. Spaseski, “A survey on model-based testing tools for test case generation,” in *Tools and Methods of Program Analysis, 2017*. Springer, 2018, pp. 77–89.

[32] I. Hooda and R. Chhillar, “A review: study of test case generation techniques,” *International Journal of Computer Applications*, vol. 107, no. 16, pp. 33–37, 2014.

[33] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann, “Test generation and test prioritization for simulink models with dynamic behavior,” *IEEE Trans. Software Eng.*, vol. 45, no. 9, pp. 919–944, 2019.

[34] C. D. Nguyen, A. Marchetto, and P. Tonella, “Combining model-based and combinatorial testing for effective test case generation,” in *International Symposium on Software Testing and Analysis*, 2012, pp. 100–110.

[35] O. Sam and N. Shankar, *Symbolic Analysis Laboratory*, accessed: 2023-10-17. [Online]. Available: https://sal.csl.sri.com/

[36] J. Li, B. Zhao, and C. Zhang, “Fuzzing: a survey,” *Cybersecur.*, vol. 1, no. 1, p. 6, 2018.

[37] P. Koopman and T. Chakravarty, “Cyclic redundancy code (CRC) polynomial selection for embedded networks,” in *International Conference on Dependable Systems and Networks*, 2004, pp. 145–154.

[38] The MathWorks Inc., “Simulink signal builder,” https://www.mathworks.com/help/simulink/ug/working-with-signal-groups.html, accessed: 2024-03-29.

[39] W. Aldrich, “Using model coverage analysis to improve the controls development process,” in *AIAA Modeling and Simulation Technologies Conference and Exhibit*, 2002, pp. 4684–4694.

[40] MathWorks, “Simulink coverage,” https://www.mathworks.com/help/slcoverage/ug/types-of-model-coverage.html, accessed: 2024-03-29.

[41] Google, “Libfuzzer tutorial,” https://github.com/google/fuzzing/blob/master/tutorial/libFuzzerTutorial.md, 2016, accessed: 2024-03-29.

[42] P. Chen and H. Chen, “Angora: Efficient fuzzing by principled search,” in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2018, pp. 711–725.

[43] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, “Driller: Augmenting fuzzing through selective symbolic execution,” in *Annual Network and Distributed System Security Symposium*, 2016.

[44] S. L. Shrestha, S. A. Chowdhury, and C. Csallner, “SLNET: A redistributable corpus of 3rd-party simulink models,” in *International Conference on Mining Software Repositories*, 2022, pp. 1–5.

[45] L. M. de Moura and N. S. Bjørner, “Z3: an efficient SMT solver,” in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, vol. 4963, 2008, pp. 337–340.

[46] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, “CVC4,” in *Computer Aided Verification - 23rd International Conference*, ser. Lecture Notes in Computer Science, vol. 6806, 2011, pp. 171–177.

[47] G. J. Holzmann, “The model checker SPIN,” *IEEE Trans. Software Eng.*, vol. 23, no. 5, pp. 279–295, 1997.

[48] K. L. McMillan, *Symbolic model checking*. Kluwer, 1993.